# Redis 中文文档

wizardforcel

Published
with GitBook

# 目錄

# Redis 中文文档

译者：黄健宏（huangz）

来源：Redis 命令参考

本文档是 Redis Command Reference 和 Redis Documentation 的中文翻译版， 阅读这个文档可以帮助你了解 Redis 命令的具体使用方法， 并学会如何使用 Redis 的事务、持久化、复制、Sentinel、集群等功能。

> 由本文档译者制作的《Redis命令速查表》正在销售中！ 该表能够与本文档相辅相成，帮助读者更好地了解和查阅 Redis 命令，有兴趣的读者可以通过访问以下链接来了解更多信息： https://selfstore.io/products/538

# Redis 文档

# 键空间通知（keyspace notification））

Note

本文档翻译自： http://redis.io/topics/notifications 。

Warning

键空间通知功能目前仍在开发中，这个文档所描述的内容，以及功能的具体实现，可能会在未来数周内改变，敬请知悉。

## 功能概览

键空间通知使得客户端可以通过订阅频道或模式， 来接收那些以某种方式改动了 Redis 数据集的事件。

以下是一些键空间通知发送的事件的例子：

- 所有修改键的命令。
- 所有接收到 LPUSH 命令的键。
- `0` 号数据库中所有已过期的键。

事件通过 Redis 的订阅与发布功能（pub/sub）来进行分发， 因此所有支持订阅与发布功能的客户端都可以在无须做任何修改的情况下， 直接使用键空间通知功能。

因为 Redis 目前的订阅与发布功能采取的是发送即忘（fire and forget）策略， 所以如果你的程序需要可靠事件通知（reliable notification of events）， 那么目前的键空间通知可能并不适合你： 当订阅事件的客户端断线时， 它会丢失所有在断线期间分发给它的事件。

未来将会支持更可靠的事件分发， 这种支持可能会通过让订阅与发布功能本身变得更可靠来实现， 也可能会在 Lua 脚本中对消息（message）的订阅与发布进行监听， 从而实现类似将事件推入到列表这样的操作。

## 事件的类型

对于每个修改数据库的操作，键空间通知都会发送两种不同类型的事件。

比如说，对 `0` 号数据库的键 `mykey` 执行 DEL 命令时， 系统将分发两条消息， 相当于执行以下两个 PUBLISH 命令：

```
PUBLISH __keyspace@0__:mykey del
PUBLISH __keyevent@0__:del mykey
```

订阅第一个频道 `__keyspace@0__:mykey` 可以接收 `0` 号数据库中所有修改键 `mykey` 的事件， 而订阅第二个频道 `__keyevent@0__:del` 则可以接收 `0` 号数据库中所有执行 `del` 命令的键。

以 `keyspace` 为前缀的频道被称为键空间通知（key-space notification）， 而以 `keyevent` 为前缀的频道则被称为键事件通知（key-event notification）。

当 `del mykey` 命令执行时：

- 键空间频道的订阅者将接收到被执行的事件的名字，在这个例子中，就是 `del` 。
- 键事件频道的订阅者将接收到被执行事件的键的名字，在这个例子中，就是 `mykey` 。

# 配置

因为开启键空间通知功能需要消耗一些 CPU ， 所以在默认配置下， 该功能处于关闭状态。

可以通过修改 `redis.conf` 文件， 或者直接使用 `CONFIG SET` 命令来开启或关闭键空间通知功能：

- 当 `notify-keyspace-events` 选项的参数为空字符串时，功能关闭。
- 另一方面，当参数不是空字符串时，功能开启。

`notify-keyspace-events` 的参数可以是以下字符的任意组合， 它指定了服务器该发送哪些类型的通知：

| 字符 | 发送的通知 |
| --- | --- |
| `K` | 键空间通知，所有通知以 `__keyspace@<db>__` 为前缀 |
| `E` | 键事件通知，所有通知以 `__keyevent@<db>__` 为前缀 |
| `g` | `DEL` 、 `EXPIRE` 、 `RENAME` 等类型无关的通用命令的通知 |
| `$` | 字符串命令的通知 |
| `l` | 列表命令的通知 |
| `s` | 集合命令的通知 |
| `h` | 哈希命令的通知 |
| `z` | 有序集合命令的通知 |
| `x` | 过期事件：每当有过期键被删除时发送 |
| `e` | 驱逐(evict)事件：每当有键因为 `maxmemory` 政策而被删除时发送 |
| `A` | 参数 `g$lshzxe` 的别名 |

输入的参数中至少要有一个 `K` 或者 `E` ， 否则的话， 不管其余的参数是什么， 都不会有任何通知被分发。

举个例子， 如果只想订阅键空间中和列表相关的通知， 那么参数就应该设为 `Kl` ， 诸如此类。

将参数设为字符串 `"AKE"` 表示发送所有类型的通知。

# 命令产生的通知

以下列表记录了不同命令所产生的不同通知：

- DEL 命令为每个被删除的键产生一个 `del` 通知。
- RENAME 产生两个通知：为来源键（source key）产生一个 `rename_from` 通知，并为目标键（destination key）产生一个 `rename_to` 通知。
- EXPIRE 和 EXPIREAT 在键被正确设置过期时间时产生一个 `expire` 通知。当 EXPIREAT 设置的时间已经过期，或者 EXPIRE 传入的时间为负数值时，键被删除，并产生一个 `del` 通知。
- SORT 在命令带有 `STORE` 参数时产生一个 `sortstore` 事件。如果 `STORE` 指示的用于保存排序结果的键已经存在，那么程序还会发送一个 `del` 事件。
- SET 以及它的所有变种（SETEX 、 SETNX 和 GETSET）都产生 `set` 通知。其中 SETEX 还会产生 `expire` 通知。
- MSET 为每个键产生一个 `set` 通知。
- SETRANGE 产生一个 `setrange` 通知。
- INCR 、 DECR 、 INCRBY 和 DECRBY 都产生 `incrby` 通知。
- INCRBYFLOAT 产生 `incrbyfloat` 通知。
- APPEND 产生 `append` 通知。
- LPUSH 和 LPUSHX 都产生单个 `lpush` 通知，即使有多个输入元素时，也是如此。
- RPUSH 和 RPUSHX 都产生单个 `rpush` 通知，即使有多个输入元素时，也是如此。
- RPOP 产生 `rpop` 通知。如果被弹出的元素是列表的最后一个元素，那么还会产生一个 `del` 通知。
- LPOP 产生 `lpop` 通知。如果被弹出的元素是列表的最后一个元素，那么还会产生一个 `del` 通知。
- LINSERT 产生一个 `linsert` 通知。
- LSET 产生一个 `lset` 通知。
- LTRIM 产生一个 `ltrim` 通知。如果 LTRIM 执行之后，列表键被清空，那么还会产生一个 `del` 通知。
- RPOPLPUSH 和 BRPOPLPUSH 产生一个 `rpop` 通知，以及一个 `lpush` 通知。两个命令都会保证 `rpop` 的通知在 `lpush` 的通知之前分发。如果从键弹出元素之后，被弹出的列表键被清空，那么还会产生一个 `del` 通知。
- HSET 、 HSETNX 和 HMSET 都只产生一个 `hset` 通知。
- HINCRBY 产生一个 `hincrby` 通知。
- HINCRBYFLOAT 产生一个 `hincrbyfloat` 通知。
- HDEL 产生一个 `hdel` 通知。如果执行 HDEL 之后，哈希键被清空，那么还会产生一个

`del` 通知。
- SADD 产生一个 `sadd` 通知，即使有多个输入元素时，也是如此。
- SREM 产生一个 `srem` 通知，如果执行 SREM 之后，集合键被清空，那么还会产生一个 `del` 通知。
- SMOVE 为来源键（source key）产生一个 `srem` 通知，并为目标键（destination key）产生一个 `sadd` 事件。
- SPOP 产生一个 `spop` 事件。如果执行 SPOP 之后，集合键被清空，那么还会产生一个 `del` 通知。
- SINTERSTORE 、 SUNIONSTORE 和 SDIFFSTORE 分别产生 `sinterstore` 、 `sunionostore` 和 `sdiffstore` 三种通知。如果用于保存结果的键已经存在，那么还会产生一个 `del` 通知。
- ZINCRBY 产生一个 `zincr` 通知。（译注：非对称，请注意。）
- ZADD 产生一个 `zadd` 通知，即使有多个输入元素时，也是如此。
- ZREM 产生一个 `zrem` 通知，即使有多个输入元素时，也是如此。如果执行 ZREM 之后，有序集合键被清空，那么还会产生一个 `del` 通知。
- ZREMRANGEBYSCORE 产生一个 `zrembyscore` 通知。（译注：非对称，请注意。）如果用于保存结果的键已经存在，那么还会产生一个 `del` 通知。
- ZREMRANGEBYRANK 产生一个 `zrembyrank` 通知。（译注：非对称，请注意。）如果用于保存结果的键已经存在，那么还会产生一个 `del` 通知。
- ZINTERSTORE 和 ZUNIONSTORE 分别产生 `zinterstore` 和 `zunionstore` 两种通知。如果用于保存结果的键已经存在，那么还会产生一个 `del` 通知。
- 每当一个键因为过期而被删除时，产生一个 `expired` 通知。
- 每当一个键因为 `maxmemory` 政策而被删除以回收内存时，产生一个 `evicted` 通知。

Note

所有命令都只在键真的被改动了之后，才会产生通知。

比如说，当 SREM 试图删除不存在于集合的元素时，删除操作会执行失败，因为没有真正的改动键，所以这一操作不会发送通知。

如果对命令所产生的通知有疑问， 最好还是使用以下命令， 自己来验证一下：

```
$ redis-cli config set notify-keyspace-events KEA
$ redis-cli --csv psubscribe '__key*__:*'
Reading messages... (press Ctrl-C to quit)
"psubscribe","__key*__:*",1
```

然后， 只要在其他终端里用 Redis 客户端发送命令， 就可以看到产生的通知了：

```
"pmessage","__key*__:*","__keyspace@0__:foo","set"
"pmessage","__key*__:*","__keyevent@0__:set","foo"
...
```

# 过期通知的发送时间

Redis 使用以下两种方式删除过期的键：

- 当一个键被访问时，程序会对这个键进行检查，如果键已经过期，那么该键将被删除。
- 底层系统会在后台渐进地查找并删除那些过期的键，从而处理那些已经过期、但是不会被访问到的键。

当过期键被以上两个程序的任意一个发现、 并且将键从数据库中删除时， Redis 会产生一个 `expired` 通知。

Redis 并不保证生存时间（TTL）变为 `0` 的键会立即被删除： 如果程序没有访问这个过期键， 或者带有生存时间的键非常多的话， 那么在键的生存时间变为 `0` ， 直到键真正被删除这中间， 可能会有一段比较显著的时间间隔。

因此， Redis 产生 `expired` 通知的时间为过期键被删除的时候， 而不是键的生存时间变为 `0` 的时候。

# 事务（transaction）

Note

本文档翻译自： http://redis.io/topics/transactions 。

MULTI 、 EXEC 、 DISCARD 和 WATCH 是 Redis 事务的基础。

事务可以一次执行多个命令， 并且带有以下两个重要的保证：

- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

  EXEC 命令负责触发并执行事务中的所有命令：

  - 如果客户端在使用 MULTI 开启了一个事务之后，却因为断线而没有成功执行 EXEC ，那么事务中的所有命令都不会被执行。
  - 另一方面，如果客户端成功在开启事务之后执行 EXEC ，那么事务中的所有命令都会被执行。

  当使用 AOF 方式做持久化的时候， Redis 会使用单个 `write(2)` 命令将事务写入到磁盘中。

  然而，如果 Redis 服务器因为某些原因被管理员杀死，或者遇上某种硬件故障，那么可能只有部分事务命令会被成功写入到磁盘中。

  如果 Redis 在重新启动时发现 AOF 文件出了这样的问题，那么它会退出，并汇报一个错误。

  使用 `redis-check-aof` 程序可以修复这一问题：它会移除 AOF 文件中不完整事务的信息，确保服务器可以顺利启动。

从 2.2 版本开始，Redis 还可以通过乐观锁（optimistic lock）实现 CAS （check-and-set）操作，具体信息请参考文档的后半部分。

# 用法

MULTI 命令用于开启一个事务，它总是返回 `OK` 。

MULTI 执行之后， 客户端可以继续向服务器发送任意多条命令， 这些命令不会立即被执行，而是被放到一个队列中， 当 EXEC 命令被调用时， 所有队列中的命令才会被执行。

另一方面， 通过调用 DISCARD ， 客户端可以清空事务队列， 并放弃执行事务。

以下是一个事务例子， 它原子地增加了 `foo` 和 `bar` 两个键的值：

```
> MULTI
OK

> INCR foo
QUEUED

> INCR bar
QUEUED

> EXEC
1) (integer) 1
2) (integer) 1
```

EXEC 命令的回复是一个数组， 数组中的每个元素都是执行事务中的命令所产生的回复。 其中， 回复元素的先后顺序和命令发送的先后顺序一致。

当客户端处于事务状态时， 所有传入的命令都会返回一个内容为 `QUEUED` 的状态回复（status reply）， 这些被入队的命令将在 EXEC 命令被调用时执行。

# 事务中的错误

使用事务时可能会遇上以下两种错误：

- 事务在执行 EXEC 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），或者其他更严重的错误，比如内存不足（如果服务器使用 `maxmemory` 设置了最大内存限制的话）。
- 命令可能在 EXEC 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。

对于发生在 EXEC 执行之前的错误，客户端以前的做法是检查命令入队所得的返回值：如果命令入队时返回 `QUEUED` ，那么入队成功；否则，就是入队失败。如果有命令在入队时失败，那么大部分客户端都会停止并取消这个事务。

不过，从 Redis 2.6.5 开始，服务器会对命令入队失败的情况进行记录，并在客户端调用 EXEC 命令时，拒绝执行并自动放弃这个事务。

在 Redis 2.6.5 以前， Redis 只执行事务中那些入队成功的命令，而忽略那些入队失败的命令。 而新的处理方式则使得在流水线（pipeline）中包含事务变得简单，因为发送事务和读取事务的回复都只需要和服务器进行一次通讯。

至于那些在 EXEC 命令执行之后所产生的错误， 并没有对它们进行特别处理： 即使事务中有某个/某些命令在执行时产生了错误， 事务中的其他命令仍然会继续执行。

从协议的角度来看这个问题，会更容易理解一些。 以下例子中， LPOP 命令的执行将出错，尽管调用它的语法是正确的：

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

MULTI
+OK

SET a 3
abc

+QUEUED
LPOP a

+QUEUED
EXEC

*2
+OK
-ERR Operation against a key holding the wrong kind of value
```

EXEC 返回两条批量回复（bulk reply）：第一条是 `OK` ，而第二条是 `-ERR` 。至于怎样用合适的方法来表示事务中的错误， 则是由客户端自己决定的。

最重要的是记住这样一条， 即使事务中有某条/某些命令执行失败了， 事务队列中的其他命令仍然会继续执行 —— Redis 不会停止执行事务中的命令。

以下例子展示的是另一种情况， 当命令在入队时产生错误， 错误会立即被返回给客户端：

```
MULTI
+OK

INCR a b c
-ERR wrong number of arguments for 'incr' command
```

因为调用 INCR 命令的参数格式不正确， 所以这个 INCR 命令入队失败。

# 为什么 Redis 不支持回滚（roll back）

如果你有使用关系式数据库的经验， 那么 “Redis 在事务失败时不进行回滚，而是继续执行余下的命令”这种做法可能会让你觉得有点奇怪。

以下是这种做法的优点：

- Redis 命令只会因为错误的语法而失败（并且这些问题不能在入队时发现），或是命令用在了错误类型的键上面：这也就是说，从实用性的角度来说，失败的命令是由编程错误造成的，而这些错误应该在开发的过程中被发现，而不应该出现在生产环境中。
- 因为不需要对回滚进行支持，所以 Redis 的内部可以保持简单且快速。

有种观点认为 Redis 处理事务的做法会产生 bug ， 然而需要注意的是， 在通常情况下， 回滚并不能解决编程错误带来的问题。 举个例子， 如果你本来想通过 INCR 命令将键的值加上 `1` ， 却不小心加上了 `2` ， 又或者对错误类型的键执行了 INCR ， 回滚是没有办法处理这

些情况的。

鉴于没有任何机制能避免程序员自己造成的错误， 并且这类错误通常不会在生产环境中出现， 所以 Redis 选择了更简单、更快速的无回滚方式来处理事务。

# 放弃事务

当执行 DISCARD 命令时， 事务会被放弃， 事务队列会被清空， 并且客户端会从事务状态中退出：

```
redis> SET foo 1
OK

redis> MULTI
OK

redis> INCR foo
QUEUED

redis> DISCARD
OK

redis> GET foo
"1"
```

# 使用 check-and-set 操作实现乐观锁

WATCH 命令可以为 Redis 事务提供 check-and-set （CAS）行为。

被 WATCH 的键会被监视，并会发觉这些键是否被改动过了。 如果有至少一个被监视的键在 EXEC 执行之前被修改了， 那么整个事务都会被取消， EXEC 返回空多条批量回复（null multi-bulk reply）来表示事务已经失败。

举个例子， 假设我们需要原子性地为某个值进行增 `1` 操作（假设 INCR 不存在）。

首先我们可能会这样做：

```
val = GET mykey
val = val + 1
SET mykey $val
```

上面的这个实现在只有一个客户端的时候可以执行得很好。 但是， 当多个客户端同时对同一个键进行这样的操作时， 就会产生竞争条件。

举个例子， 如果客户端 A 和 B 都读取了键原来的值， 比如 `10` ， 那么两个客户端都会将键的值设为 `11` ， 但正确的结果应该是 `12` 才对。

有了 WATCH ， 我们就可以轻松地解决这类问题了：

```
WATCH mykey

val = GET mykey
val = val + 1

MULTI
SET mykey $val
EXEC
```

使用上面的代码， 如果在 WATCH 执行之后， EXEC 执行之前， 有其他客户端修改了 `mykey` 的值， 那么当前客户端的事务就会失败。 程序需要做的， 就是不断重试这个操作，直到没有发生碰撞为止。

这种形式的锁被称作乐观锁， 它是一种非常强大的锁机制。 并且因为大多数情况下， 不同的客户端会访问不同的键， 碰撞的情况一般都很少， 所以通常并不需要进行重试。

# 了解 WATCH

WATCH 使得 EXEC 命令需要有条件地执行： 事务只能在所有被监视键都没有被修改的前提下执行， 如果这个前提不能满足的话，事务就不会被执行。

Note

如果你使用 WATCH 监视了一个带过期时间的键， 那么即使这个键过期了， 事务仍然可以正常执行， 关于这方面的详细情况，请看这个帖子：http://code.google.com/p/redis/issues/detail?id=270

WATCH 命令可以被调用多次。 对键的监视从 WATCH 执行之后开始生效， 直到调用 EXEC 为止。

用户还可以在单个 WATCH 命令中监视任意多个键， 就像这样：

```
redis> WATCH key1 key2 key3
OK
```

当 EXEC 被调用时， 不管事务是否成功执行， 对所有键的监视都会被取消。

另外， 当客户端断开连接时， 该客户端对键的监视也会被取消。

使用无参数的 UNWATCH 命令可以手动取消对所有键的监视。 对于一些需要改动多个键的事务， 有时候程序需要同时对多个键进行加锁， 然后检查这些键的当前值是否符合程序的要求。 当值达不到要求时， 就可以使用 UNWATCH 命令来取消目前对键的监视， 中途放弃这个事务， 并等待事务的下次尝试。

# 使用 WATCH 实现 ZPOP

WATCH 可以用于创建 Redis 没有内置的原子操作。

举个例子， 以下代码实现了原创的 `ZPOP` 命令， 它可以原子地弹出有序集合中分值（score）最小的元素：

```
WATCH zset
element = ZRANGE zset 0 0
MULTI
    ZREM zset element
EXEC
```

程序只要重复执行这段代码， 直到 EXEC 的返回值不是空多条回复（null multi-bulk reply）即可。

## Redis 脚本和事务

从定义上来说， Redis 中的脚本本身就是一种事务， 所以任何在事务里可以完成的事， 在脚本里面也能完成。 并且一般来说， 使用脚本要来得更简单，并且速度更快。

因为脚本功能是 Redis 2.6 才引入的， 而事务功能则更早之前就存在了， 所以 Redis 才会同时存在两种处理事务的方法。

不过我们并不打算在短时间内就移除事务功能， 因为事务提供了一种即使不使用脚本， 也可以避免竞争条件的方法， 而且事务本身的实现并不复杂。

不过在不远的将来， 可能所有用户都会只使用脚本来实现事务也说不定。 如果真的发生这种情况的话， 那么我们将废弃并最终移除事务功能。

# 发布与订阅（pub/sub）

Note

本文档翻译自： http://redis.io/topics/pubsub 。

SUBSCRIBE 、 UNSUBSCRIBE 和 PUBLISH 三个命令实现了发布与订阅信息泛型（Publish/Subscribe messaging paradigm）， 在这个实现中， 发送者（发送信息的客户端）不是将信息直接发送给特定的接收者（接收信息的客户端）， 而是将信息发送给频道（channel）， 然后由频道将信息转发给所有对这个频道感兴趣的订阅者。

发送者无须知道任何关于订阅者的信息， 而订阅者也无须知道是那个客户端给它发送信息， 它只要关注自己感兴趣的频道即可。

对发布者和订阅者进行解构（decoupling）， 可以极大地提高系统的扩展性（scalability）， 并得到一个更动态的网络拓扑（network topology）。

比如说， 要订阅频道 `foo` 和 `bar` ， 客户端可以使用频道名字作为参数来调用 SUBSCRIBE 命令：

```
redis> SUBSCRIBE foo bar
```

当有客户端发送信息到这些频道时， Redis 会将传入的信息推送到所有订阅这些频道的客户端里面。

正在订阅频道的客户端不应该发送除 SUBSCRIBE 和 UNSUBSCRIBE 之外的其他命令。 其中， SUBSCRIBE 可以用于订阅更多频道， 而 UNSUBSCRIBE 则可以用于退订已订阅的一个或多个频道。

SUBSCRIBE 和 UNSUBSCRIBE 的执行结果会以信息的形式返回， 客户端可以通过分析所接收信息的第一个元素， 从而判断所收到的内容是一条真正的信息， 还是 SUBSCRIBE 或 UNSUBSCRIBE 命令的操作结果。

## 信息的格式

频道转发的每条信息都是一条带有三个元素的多条批量回复（multi-bulk reply）。

信息的第一个元素标识了信息的类型：

- `subscribe` ： 表示当前客户端成功地订阅了信息第二个元素所指示的频道。 而信息的第三个元素则记录了目前客户端已订阅频道的总数。
- `unsubscribe` ： 表示当前客户端成功地退订了信息第二个元素所指示的频道。 信息的第

三个元素记录了客户端目前仍在订阅的频道数量。 当客户端订阅的频道数量降为 `0` 时， 客户端不再订阅任何频道， 它可以像往常一样， 执行任何 Redis 命令。
- `message` ： 表示这条信息是由某个客户端执行 PUBLISH 命令所发送的， 真正的信息。 信息的第二个元素是信息来源的频道， 而第三个元素则是信息的内容。

举个例子， 如果客户端执行以下命令：

```
redis> SUBSCRIBE first second
```

那么它将收到以下回复：

```
1) "subscribe"
2) "first"
3) (integer) 1

1) "subscribe"
2) "second"
3) (integer) 2
```

如果在这时， 另一个客户端执行以下 PUBLISH 命令：

```
redis> PUBLISH second Hello
```

那么之前订阅了 `second` 频道的客户端将收到以下信息：

```
1) "message"
2) "second"
3) "hello"
```

当订阅者决定退订所有频道时， 它可以执行一个无参数的 UNSUBSCRIBE 命令：

```
redis> UNSUBSCRIBE
```

这个命令将接到以下回复：

```
1) "unsubscribe"
2) "second"
3) (integer) 1

1) "unsubscribe"
2) "first"
3) (integer) 0
```

# 订阅模式

Redis 的发布与订阅实现支持模式匹配（pattern matching）： 客户端可以订阅一个带 `*` 号的模式， 如果某个/某些频道的名字和这个模式匹配， 那么当有信息发送给这个/这些频道的时候， 客户端也会收到这个/这些频道的信息。

比如说，执行命令

```
redis> PSUBSCRIBE news.*
```

的客户端将收到来自 `news.art.figurative` 、 `news.music.jazz` 等频道的信息。

客户端订阅的模式里面可以包含多个 glob 风格的通配符， 比如 `*` 、 `?` 和 `[...]` ， 等等。

执行命令

```
redis> PUNSUBSCRIBE news.*
```

将退订 `news.*` 模式， 其他已订阅的模式不会被影响。

通过订阅模式接收到的信息， 和通过订阅频道接收到的信息， 这两者的格式不太一样：

- 通过订阅模式而接收到的信息的类型为 `pmessage` ： 这代表有某个客户端通过 PUBLISH 向某个频道发送了信息， 而这个频道刚好匹配了当前客户端所订阅的某个模式。 信息的第二个元素记录了被匹配的模式， 第三个元素记录了被匹配的频道的名字， 最后一个元素则记录了信息的实际内容。

客户端处理 PSUBSCRIBE 和 PUNSUBSCRIBE 返回值的方式， 和客户端处理 SUBSCRIBE 和 UNSUBSCRIBE 的方式类似： 通过对信息的第一个元素进行分析， 客户端可以判断接收到的信息是一个真正的信息， 还是 PSUBSCRIBE 或 PUNSUBSCRIBE 命令的返回值。

## 通过频道和模式接收同一条信息

如果客户端订阅的多个模式匹配了同一个频道， 或者客户端同时订阅了某个频道、以及匹配这个频道的某个模式， 那么它可能会多次接收到同一条信息。

举个例子， 如果客户端执行了以下命令：

```
SUBSCRIBE foo
PSUBSCRIBE f*
```

那么当有信息发送到频道 `foo` 时， 客户端将收到两条信息： 一条来自频道 `foo` ， 信息类型为 `message` ； 另一条来自模式 `f*` ， 信息类型为 `pmessage` 。

# 订阅总数

在执行 SUBSCRIBE 、 UNSUBSCRIBE 、 PSUBSCRIBE 和 PUNSUBSCRIBE 命令时， 返回结果的最后一个元素是客户端目前仍在订阅的频道和模式总数。

当客户端退订所有频道和模式， 也即是这个总数值下降为 `0` 的时候， 客户端将退出订阅与发布状态。

# 编程示例

Pieter Noordhuis 提供了一个使用 EventMachine 和 Redis 编写的 高性能多用户网页聊天软件 ， 这个软件很好地展示了发布与订阅功能的用法。

# 客户端库实现提示

因为所有接收到的信息都会包含一个信息来源：

- 当信息来自频道时，来源是某个频道；
- 当信息来自模式时，来源是某个模式。

因此， 客户端可以用一个哈希表， 将特定来源和处理该来源的回调函数关联起来。 当有新信息到达时， 程序就可以根据信息的来源， 在 O(1) 复杂度内， 将信息交给正确的回调函数来处理。

# 复制（Replication）

Note

本文档翻译自： http://redis.io/topics/replication 。

Redis 支持简单且易用的主从复制（master-slave replication）功能， 该功能可以让从服务器(slave server)成为主服务器(master server)的精确复制品。

以下是关于 Redis 复制功能的几个重要方面：

- Redis 使用异步复制。 从 Redis 2.8 开始， 从服务器会以每秒一次的频率向主服务器报告复制流（replication stream）的处理进度。
- 一个主服务器可以有多个从服务器。
- 不仅主服务器可以有从服务器， 从服务器也可以有自己的从服务器， 多个从服务器之间可以构成一个图状结构。
- 复制功能不会阻塞主服务器： 即使有一个或多个从服务器正在进行初次同步， 主服务器也可以继续处理命令请求。
- 复制功能也不会阻塞从服务器： 只要在 `redis.conf` 文件中进行了相应的设置， 即使从服务器正在进行初次同步， 服务器也可以使用旧版本的数据集来处理命令查询。

  不过， 在从服务器删除旧版本数据集并载入新版本数据集的那段时间内， 连接请求会被阻塞。

  你还可以配置从服务器， 让它在与主服务器之间的连接断开时， 向客户端发送一个错误。
- 复制功能可以单纯地用于数据冗余（data redundancy）， 也可以通过让多个从服务器处理只读命令请求来提升扩展性（scalability）： 比如说， 繁重的 SORT 命令可以交给附属节点去运行。
- 可以通过复制功能来让主服务器免于执行持久化操作： 只要关闭主服务器的持久化功能， 然后由从服务器去执行持久化操作即可。

## 关闭主服务器持久化时，复制功能的数据安全

当配置Redis复制功能时，强烈建议打开主服务器的持久化功能。 否则的话，由于延迟等问题，部署的服务应该要避免自动拉起。

为了帮助理解主服务器关闭持久化时自动拉起的危险性，参考一下以下会导致主从服务器数据全部丢失的例子：

1. 假设节点A为主服务器，并且关闭了持久化。 并且节点B和节点C从节点A复制数据

2. 节点A崩溃，然后由自动拉起服务重启了节点A. 由于节点A的持久化被关闭了，所以重启之后没有任何数据

3. 节点B和节点C将从节点A复制数据，但是A的数据是空的， 于是就把自身保存的数据副本删除。

在关闭主服务器上的持久化，并同时开启自动拉起进程的情况下，即便使用Sentinel来实现Redis的高可用性，也是非常危险的。 因为主服务器可能拉起得非常快，以至于Sentinel在配置的心跳时间间隔内没有检测到主服务器已被重启，然后还是会执行上面的数据丢失的流程。

无论何时，数据安全都是极其重要的，所以应该禁止主服务器关闭持久化的同时自动拉起。

# 复制功能的运作原理

无论是初次连接还是重新连接， 当建立一个从服务器时， 从服务器都将向主服务器发送一个 SYNC 命令。

接到 SYNC 命令的主服务器将开始执行 BGSAVE ， 并在保存操作执行期间， 将所有新执行的写入命令都保存到一个缓冲区里面。

当 BGSAVE 执行完毕后， 主服务器将执行保存操作所得的 `.rdb` 文件发送给从服务器， 从服务器接收这个 `.rdb` 文件， 并将文件中的数据载入到内存中。

之后主服务器会以 Redis 命令协议的格式， 将写命令缓冲区中积累的所有内容都发送给从服务器。

你可以通过 telnet 命令来亲自验证这个同步过程： 首先连上一个正在处理命令请求的 Redis 服务器， 然后向它发送 SYNC 命令， 过一阵子， 你将看到 telnet 会话（session）接收到服务器发来的大段数据（ `.rdb` 文件）， 之后还会看到， 所有在服务器执行过的写命令， 都会重新发送到 telnet 会话来。

即使有多个从服务器同时向主服务器发送 SYNC ， 主服务器也只需执行一次 BGSAVE 命令， 就可以处理所有这些从服务器的同步请求。

从服务器可以在主从服务器之间的连接断开时进行自动重连， 在 Redis 2.8 版本之前， 断线之后重连的从服务器总要执行一次完整重同步（full resynchronization）操作， 但是从 Redis 2.8 版本开始， 从服务器可以根据主服务器的情况来选择执行完整重同步还是部分重同步（partial resynchronization）。

# 部分重同步

从 Redis 2.8 开始， 在网络连接短暂性失效之后， 主从服务器可以尝试继续执行原有的复制进程（process）， 而不一定要执行完整重同步操作。

这个特性需要主服务器为被发送的复制流创建一个内存缓冲区（in-memory backlog）， 并且主服务器和所有从服务器之间都记录一个复制偏移量（replication offset）和一个主服务器 ID（master run id）， 当出现网络连接断开时， 从服务器会重新连接， 并且向主服务器请求继续执行原来的复制进程：

- 如果从服务器记录的主服务器 ID 和当前要连接的主服务器的 ID 相同， 并且从服务器记录的偏移量所指定的数据仍然保存在主服务器的复制流缓冲区里面， 那么主服务器会向从服务器发送断线时缺失的那部分数据， 然后复制工作可以继续执行。
- 否则的话， 从服务器就要执行完整重同步操作。

Redis 2.8 的这个部分重同步特性会用到一个新增的 PSYNC 内部命令， 而 Redis 2.8 以前的旧版本只有 SYNC 命令， 不过， 只要从服务器是 Redis 2.8 或以上的版本， 它就会根据主服务器的版本来决定到底是使用 PSYNC 还是 SYNC ：

- 如果主服务器是 Redis 2.8 或以上版本，那么从服务器使用 PSYNC 命令来进行同步。
- 如果主服务器是 Redis 2.8 之前的版本，那么从服务器使用 SYNC 命令来进行同步。

# 配置

配置一个从服务器非常简单， 只要在配置文件中增加以下的这一行就可以了：

```
slaveof 192.168.1.1 6379
```

当然， 你需要将代码中的 `192.168.1.1` 和 `6379` 替换成你的主服务器的 IP 和端口号。

另外一种方法是调用 SLAVEOF 命令， 输入主服务器的 IP 和端口， 然后同步就会开始：

```
127.0.0.1:6379> SLAVEOF 192.168.1.1 10086
OK
```

# 只读从服务器

从 Redis 2.6 开始， 从服务器支持只读模式， 并且该模式为从服务器的默认模式。

只读模式由 `redis.conf` 文件中的 `slave-read-only` 选项控制， 也可以通过 CONFIG SET 命令来开启或关闭这个模式。

只读从服务器会拒绝执行任何写命令， 所以不会出现因为操作失误而将数据不小心写入到了从服务器的情况。

即使从服务器是只读的， `DEBUG` 和 `CONFIG` 等管理式命令仍然是可以使用的， 所以我们还是不应该将服务器暴露给互联网或者任何不可信网络。 不过， 使用 `redis.conf` 中的命令改名选项， 我们可以通过禁止执行某些命令来提升只读从服务器的安全性。

你可能会感到好奇， 既然从服务器上的写数据会被重同步数据覆盖， 也可能在从服务器重启时丢失， 那么为什么要让一个从服务器变得可写呢？

原因是， 一些不重要的临时数据， 仍然是可以保存在从服务器上面的。 比如说， 客户端可以在从服务器上保存主服务器的可达性（reachability）信息， 从而实现故障转移（failover）策略。

# 从服务器相关配置

如果主服务器通过 `requirepass` 选项设置了密码， 那么为了让从服务器的同步操作可以顺利进行， 我们也必须为从服务器进行相应的身份验证设置。

对于一个正在运行的服务器， 可以使用客户端输入以下命令：

```
config set masterauth <password>
```

要永久地设置这个密码， 那么可以将它加入到配置文件中：

```
masterauth <password>
```

另外还有几个选项， 它们和主服务器执行部分重同步时所使用的复制流缓冲区有关， 详细的信息可以参考 Redis 源码中附带的 `redis.conf` 示例文件。

# 主服务器只在有至少 N 个从服务器的情况下，才执行写操作

从 Redis 2.8 开始， 为了保证数据的安全性， 可以通过配置， 让主服务器只在有至少 N 个当前已连接从服务器的情况下， 才执行写命令。

不过， 因为 Redis 使用异步复制， 所以主服务器发送的写数据并不一定会被从服务器接收到， 因此， 数据丢失的可能性仍然是存在的。

以下是这个特性的运作原理：

- 从服务器以每秒一次的频率 PING 主服务器一次， 并报告复制流的处理情况。
- 主服务器会记录各个从服务器最后一次向它发送 PING 的时间。

- 用户可以通过配置， 指定网络延迟的最大值 `min-slaves-max-lag` ， 以及执行写操作所需的至少从服务器数量 `min-slaves-to-write` 。

如果至少有 `min-slaves-to-write` 个从服务器， 并且这些服务器的延迟值都少于 `min-slaves-max-lag` 秒， 那么主服务器就会执行客户端请求的写操作。

你可以将这个特性看作 CAP 理论中的 C 的条件放宽版本： 尽管不能保证写操作的持久性， 但起码丢失数据的窗口会被严格限制在指定的秒数中。

另一方面， 如果条件达不到 `min-slaves-to-write` 和 `min-slaves-max-lag` 所指定的条件， 那么写操作就不会被执行， 主服务器会向请求执行写操作的客户端返回一个错误。

以下是这个特性的两个选项和它们所需的参数：

- `min-slaves-to-write <number of slaves>`
- `min-slaves-max-lag <number of seconds>`

详细的信息可以参考 Redis 源码中附带的 `redis.conf` 示例文件。

# 通信协议（protocol）

Note

本文档翻译自： http://redis.io/topics/protocol 。

Redis 协议在以下三个目标之间进行折中：

- 易于实现
- 可以高效地被计算机分析（parse）
- 可以很容易地被人类读懂

## 网络层

客户端和服务器通过 TCP 连接来进行数据交互， 服务器默认的端口号为 6379 。

客户端和服务器发送的命令或数据一律以 `\r\n` （CRLF）结尾。

## 请求

Redis 服务器接受命令以及命令的参数。

服务器会在接到命令之后， 对命令进行处理， 并将命令的回复传送回客户端。

## 新版统一请求协议

新版统一请求协议在 Redis 1.2 版本中引入， 并最终在 Redis 2.0 版本成为 Redis 服务器通信的标准方式。

你的 Redis 客户端应该按照这个新版协议来进行实现。

在这个协议中， 所有发送至 Redis 服务器的参数都是二进制安全（binary safe）的。

以下是这个协议的一般形式：

```
*<参数数量> CR LF
$<参数 1 的字节数量> CR LF
<参数 1 的数据> CR LF
...
$<参数 N 的字节数量> CR LF
<参数 N 的数据> CR LF
```

Note

译注：命令本身也作为协议的其中一个参数来发送。

举个例子， 以下是一个命令协议的打印版本：

```
*3
$3
SET
$5
mykey
$7
myvalue
```

这个命令的实际协议值如下：

```
"*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"
```

稍后我们会看到， 这种格式除了用作命令请求协议之外， 也用在命令的回复协议中： 这种只有一个参数的回复格式被称为批量回复（**Bulk Reply**）。

统一协议请求原本是用在回复协议中， 用于将列表的多个项返回给客户端的， 这种回复格式被称为多条批量回复（**Multi Bulk Reply**）。

一个多条批量回复以 `*<argc>\r\n` 为前缀， 后跟多条不同的批量回复， 其中 `argc` 为这些批量回复的数量。

# 回复

Redis 命令会返回多种不同类型的回复。

通过检查服务器发回数据的第一个字节， 可以确定这个回复是什么类型：

- 状态回复（status reply）的第一个字节是 `"+"`
- 错误回复（error reply）的第一个字节是 `"-"`
- 整数回复（integer reply）的第一个字节是 `":"`
- 批量回复（bulk reply）的第一个字节是 `"$"`
- 多条批量回复（multi bulk reply）的第一个字节是 `"*"`

# 状态回复

一个状态回复（或者单行回复，single line reply）是一段以 `"+"` 开始、 `"\r\n"` 结尾的单行字符串。

以下是一个状态回复的例子：

```
+OK
```

客户端库应该返回 `"+"` 号之后的所有内容。 比如在在上面的这个例子中， 客户端就应该返回字符串 `"OK"` 。

状态回复通常由那些不需要返回数据的命令返回，这种回复不是二进制安全的，它也不能包含新行。

状态回复的额外开销非常少，只需要三个字节（开头的 `"+"` 和结尾的 CRLF）。

# 错误回复

错误回复和状态回复非常相似， 它们之间的唯一区别是， 错误回复的第一个字节是 `"-"` ，而状态回复的第一个字节是 `"+"` 。

错误回复只在某些地方出现问题时发送： 比如说， 当用户对不正确的数据类型执行命令， 或者执行一个不存在的命令， 等等。

一个客户端库应该在收到错误回复时产生一个异常。

以下是两个错误回复的例子：

```
-ERR unknown command 'foobar'
-WRONGTYPE Operation against a key holding the wrong kind of value
```

在 `"-"` 之后，直到遇到第一个空格或新行为止，这中间的内容表示所返回错误的类型。

`ERR` 是一个通用错误，而 `WRONGTYPE` 则是一个更特定的错误。 一个客户端实现可以为不同类型的错误产生不同类型的异常， 或者提供一种通用的方式， 让调用者可以通过提供字符串形式的错误名来捕捉（trap）不同的错误。

不过这些特性用得并不多， 所以并不是特别重要， 一个受限的（limited）客户端可以通过简单地返回一个逻辑假（false）来表示一个通用的错误条件。

# 整数回复

整数回复就是一个以 `":"` 开头， CRLF 结尾的字符串表示的整数。

比如说， `":0\r\n"` 和 `":1000\r\n"` 都是整数回复。

返回整数回复的其中两个命令是 INCR 和 LASTSAVE 。 被返回的整数没有什么特殊的含义， INCR 返回键的一个自增后的整数值， 而 LASTSAVE 则返回一个 UNIX 时间戳， 返回值的唯一限制是这些数必须能够用 64 位有符号整数表示。

整数回复也被广泛地用于表示逻辑真和逻辑假： 比如 EXISTS 和 SISMEMBER 都用返回值 `1` 表示真， `0` 表示假。

其他一些命令， 比如 SADD 、 SREM 和 SETNX ， 只在操作真正被执行了的时候， 才返回 `1` ， 否则返回 `0` 。

以下命令都返回整数回复： SETNX 、 DEL 、 EXISTS 、 INCR 、 INCRBY 、 DECR 、 DECRBY 、 DBSIZE 、 LASTSAVE 、 RENAMENX 、 MOVE 、 LLEN 、 SADD 、 SREM 、 SISMEMBER 、 SCARD 。

# 批量回复

服务器使用批量回复来返回二进制安全的字符串，字符串的最大长度为 512 MB 。

```
客户端：GET mykey
服务器：foobar
```

服务器发送的内容中：

- 第一字节为 `"$"` 符号
- 接下来跟着的是表示实际回复长度的数字值
- 之后跟着一个 CRLF
- 再后面跟着的是实际回复数据
- 最末尾是另一个 CRLF

对于前面的 GET 命令，服务器实际发送的内容为：

```
"$6\r\nfoobar\r\n"
```

如果被请求的值不存在， 那么批量回复会将特殊值 `-1` 用作回复的长度值， 就像这样：

```
客户端：GET non-existing-key
服务器：$-1
```

这种回复称为空批量回复（NULL Bulk Reply）。

当请求对象不存在时，客户端应该返回空对象，而不是空字符串： 比如 Ruby 库应该返回 `nil` ， 而 C 库应该返回 `NULL` （或者在回复对象中设置一个特殊标志）， 诸如此类。

# 多条批量回复

像 LRANGE 这样的命令需要返回多个值， 这一目标可以通过多条批量回复来完成。

多条批量回复是由多个回复组成的数组， 数组中的每个元素都可以是任意类型的回复， 包括多条批量回复本身。

多条批量回复的第一个字节为 `"*"` ， 后跟一个字符串表示的整数值， 这个值记录了多条批量回复所包含的回复数量， 再后面是一个 CRLF 。

```
客户端： LRANGE mylist 0 3
服务器： *4
服务器： $3
服务器： foo
服务器： $3
服务器： bar
服务器： $5
服务器： Hello
服务器： $5
服务器： World
```

在上面的示例中，服务器发送的所有字符串都由 CRLF 结尾。

正如你所见到的那样， 多条批量回复所使用的格式， 和客户端发送命令时使用的统一请求协议的格式一模一样。 它们之间的唯一区别是：

- 统一请求协议只发送批量回复。
- 而服务器应答命令时所发送的多条批量回复，则可以包含任意类型的回复。

以下例子展示了一个多条批量回复， 回复中包含四个整数值， 以及一个二进制安全字符串：

```
*5\r\n
:1\r\n
:2\r\n
:3\r\n
:4\r\n
$6\r\n
foobar\r\n
```

在回复的第一行， 服务器发送 `*5\r\n` ， 表示这个多条批量回复包含 5 条回复， 再后面跟着的则是 5 条回复的正文。

多条批量回复也可以是空白的（empty）， 就像这样：

```
客户端： LRANGE nokey 0 1
服务器： *0\r\n
```

无内容的多条批量回复（null multi bulk reply）也是存在的， 比如当 BLPOP 命令的阻塞时间超过最大时限时， 它就返回一个无内容的多条批量回复， 这个回复的计数值为 `-1` ：

```
客户端： BLPOP key 1
服务器： *-1\r\n
```

客户端库应该区别对待空白多条回复和无内容多条回复： 当 Redis 返回一个无内容多条回复时， 客户端库应该返回一个 null 对象， 而不是一个空数组。

# 多条批量回复中的空元素

多条批量回复中的元素可以将自身的长度设置为 `-1` ， 从而表示该元素不存在， 并且也不是一个空白字符串（empty string）。

当 SORT 命令使用 `GET pattern` 选项对一个不存在的键进行操作时， 就会发生多条批量回复中带有空白元素的情况。

以下例子展示了一个包含空元素的多重批量回复：

```
服务器： *3
服务器： $3
服务器： foo
服务器： $-1
服务器： $3
服务器： bar
```

其中， 回复中的第二个元素为空。

对于这个回复， 客户端库应该返回类似于这样的回复：

```
["foo", nil, "bar"]
```

# 多命令和流水线

客户端可以通过流水线， 在一次写入操作中发送多个命令：

- 在发送新命令之前， 无须阅读前一个命令的回复。
- 多个命令的回复会在最后一并返回。

# 内联命令

当你需要和 Redis 服务器进行沟通， 但又找不到 `redis-cli` ， 而手上只有 `telnet` 的时候， 你可以通过 Redis 特别为这种情形而设的内联命令格式来发送命令。

以下是一个客户端和服务器使用内联命令来进行交互的例子：

```
客户端： PING
服务器： +PONG
```

以下另一个返回整数值的内联命令的例子：

```
客户端： EXISTS somekey
服务器： :0
```

因为没有了统一请求协议中的 `"*"` 项来声明参数的数量， 所以在 `telnet` 会话输入命令的时候， 必须使用空格来分割各个参数， 服务器在接收到数据之后， 会按空格对用户的输入进行分析（parse）， 并获取其中的命令参数。

# 高性能 Redis 协议分析器

尽管 Redis 的协议非常利于人类阅读， 定义也很简单， 但这个协议的实现性能仍然可以和二进制协议一样快。

因为 Redis 协议将数据的长度放在数据正文之前， 所以程序无须像 JSON 那样， 为了寻找某个特殊字符而扫描整个 payload ， 也无须对发送至服务器的 payload 进行转义（quote）。

程序可以在对协议文本中的各个字符进行处理的同时， 查找 CR 字符， 并计算出批量回复或多条批量回复的长度， 就像这样：

```
#include <stdio.h>

int main(void) {
    unsigned char *p = "$123\r\n";
    int len = 0;

    p++;
    while(*p != '\r') {
        len = (len*10)+(*p - '0');
        p++;
    }

    /* Now p points at '\r', and the len is in bulk_len. */
    printf("%d\n", len);
    return 0;
}
```

得到了批量回复或多条批量回复的长度之后， 程序只需调用一次 `read` 函数， 就可以将回复的正文数据全部读入到内存中， 而无须对这些数据做任何的处理。

在回复最末尾的 CR 和 LF 不作处理，丢弃它们。

Redis 协议的实现性能可以和二进制协议的实现性能相媲美， 并且由于 Redis 协议的简单性， 大部分高级语言都可以轻易地实现这个协议， 这使得客户端软件的 bug 数量大大减少。

# 持久化（persistence）

Note

本文档翻译自 http://redis.io/topics/persistence 。

这篇文章提供了 Redis 持久化的技术性描述， 推荐所有 Redis 用户阅读。

要更广泛地了解 Redis 持久化， 以及这种持久化所保证的耐久性（durability）， 请参考文章 Redis persistence demystified （中文）。

## Redis 持久化

Redis 提供了多种不同级别的持久化方式：

- RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）。
- AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。 Redis 还可以在后台对 AOF 文件进行重写（rewrite），使得 AOF 文件的体积不会超出保存数据集状态所需的实际大小。
- Redis 还可以同时使用 AOF 持久化和 RDB 持久化。 在这种情况下， 当 Redis 重启时， 它会优先使用 AOF 文件来还原数据集， 因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。
- 你甚至可以关闭持久化功能，让数据只在服务器运行时存在。

了解 RDB 持久化和 AOF 持久化之间的异同是非常重要的， 以下几个小节将详细地介绍这这两种持久化功能， 并对它们的相同和不同之处进行说明。

## RDB 的优点

- RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。
- RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。
- RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 `fork` 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。

- RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

## RDB 的缺点

- 如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。
- 每次保存 RDB 的时候，Redis 都要 `fork()` 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， `fork()` 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 `fork()` ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

## AOF 的优点

- 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 `fsync` 策略，比如无 `fsync` ，每秒钟一次 `fsync` ，或者每次执行写入命令时 `fsync` 。 AOF 的默认策略为每秒钟 `fsync` 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ `fsync` 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
- AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 `seek` ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， `redis-check-aof` 工具也可以轻易地修复这种问题。
- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。
- AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

## AOF 的缺点

- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。
- 根据所使用的 `fsync` 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 `fsync` 的性能依然非常高， 而关闭 `fsync` 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。
- AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的。

# RDB 和 AOF ， 我应该用哪一个？

一般来说， 如果想达到足以媲美 PostgreSQL 的数据安全性， 你应该同时使用两种持久化功能。

如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久化。

有很多用户都只使用 AOF 持久化， 但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快， 除此之外， 使用 RDB 还可以避免之前提到的 AOF 程序的 bug 。

Note

因为以上提到的种种原因， 未来我们可能会将 AOF 和 RDB 整合成单个持久化模型。 （这是一个长期计划。）

接下来的几个小节将介绍 RDB 和 AOF 的更多细节。

# RDB 快照

在默认情况下， Redis 将数据库快照保存在名字为 `dump.rdb` 的二进制文件中。

你可以对 Redis 进行设置， 让它在“ `N` 秒内数据集至少有 `M` 个改动”这一条件被满足时， 自动保存一次数据集。

你也可以通过调用 SAVE 或者 BGSAVE ， 手动让 Redis 进行数据集保存操作。

比如说， 以下设置会让 Redis 在满足“ `60` 秒内有至少有 `1000` 个键被改动”这一条件时， 自动保存一次数据集：

```
save 60 1000
```

这种持久化方式被称为快照（snapshot）。

## 快照的运作方式

当 Redis 需要保存 `dump.rdb` 文件时， 服务器执行以下操作：

1. Redis 调用 `fork()` ，同时拥有父进程和子进程。
2. 子进程将数据集写入到一个临时 RDB 文件中。
3. 当子进程完成对新 RDB 文件的写入时，Redis 用新 RDB 文件替换原来的 RDB 文件，并删除旧的 RDB 文件。

这种工作方式使得 Redis 可以从写时复制（copy-on-write）机制中获益。

# 只进行追加操作的文件（append-only file，AOF）

快照功能并不是非常耐久（durable）： 如果 Redis 因为某些原因而造成故障停机， 那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。

尽管对于某些程序来说， 数据的耐久性并不是最重要的考虑因素， 但是对于那些追求完全耐久能力（full durability）的程序来说， 快照功能就不太适用了。

从 1.1 版本开始， Redis 增加了一种完全耐久的持久化方式： AOF 持久化。

你可以通过修改配置文件来打开 AOF 功能：

```
appendonly yes
```

从现在开始， 每当 Redis 执行一个改变数据集的命令时（比如 SET）， 这个命令就会被追加到 AOF 文件的末尾。

这样的话， 当 Redis 重新启时， 程序就可以通过重新执行 AOF 文件中的命令来达到重建数据集的目的。

# AOF 重写

因为 AOF 的运作方式是不断地将命令追加到文件的末尾， 所以随着写入命令的不断增加， AOF 文件的体积也会变得越来越大。

举个例子， 如果你对一个计数器调用了 100 次 INCR ， 那么仅仅是为了保存这个计数器的当前值， AOF 文件就需要使用 100 条记录（entry）。

然而在实际上， 只使用一条 SET 命令已经足以保存计数器的当前值了， 其余 99 条记录实际上都是多余的。

为了处理这种情况， Redis 支持一种有趣的特性： 可以在不打断服务客户端的情况下， 对 AOF 文件进行重建（rebuild）。

执行 BGREWRITEAOF 命令， Redis 将生成一个新的 AOF 文件， 这个文件包含重建当前数据集所需的最少命令。

Redis 2.2 需要自己手动执行 BGREWRITEAOF 命令； Redis 2.4 则可以自动触发 AOF 重写， 具体信息请查看 2.4 的示例配置文件。

# AOF 的耐久性如何？

你可以配置 Redis 多久才将数据 `fsync` 到磁盘一次。

有三个选项：

- 每次有新命令追加到 AOF 文件时就执行一次 `fsync` ：非常慢，也非常安全。
- 每秒 `fsync` 一次：足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失 1 秒钟的数据。
- 从不 `fsync` ：将数据交给操作系统来处理。更快，也更不安全的选择。

推荐（并且也是默认）的措施为每秒 `fsync` 一次， 这种 `fsync` 策略可以兼顾速度和安全性。

总是 `fsync` 的策略在实际使用中非常慢， 即使在 Redis 2.0 对相关的程序进行了改进之后仍是如此 —— 频繁调用 `fsync` 注定了这种策略不可能快得起来。

# 如果 AOF 文件出错了，怎么办？

服务器可能在程序正在对 AOF 文件进行写入时停机， 如果停机造成了 AOF 文件出错（corrupt）， 那么 Redis 在重启时会拒绝载入这个 AOF 文件， 从而确保数据的一致性不会被破坏。

当发生这种情况时， 可以用以下方法来修复出错的 AOF 文件：

1. 为现有的 AOF 文件创建一个备份。
2. 使用 Redis 附带的 `redis-check-aof` 程序，对原来的 AOF 文件进行修复。

> `> $ redis-check-aof --fix > >`

1. （可选）使用 `diff -u` 对比修复后的 AOF 文件和原始 AOF 文件的备份，查看两个文件之间的不同之处。
2. 重启 Redis 服务器，等待服务器载入修复后的 AOF 文件，并进行数据恢复。

# AOF 的运作方式

AOF 重写和 RDB 创建快照一样，都巧妙地利用了写时复制机制。

以下是 AOF 重写的执行步骤：

1. Redis 执行 `fork()` ，现在同时拥有父进程和子进程。
2. 子进程开始将新 AOF 文件的内容写入到临时文件。
3. 对于所有新执行的写入命令，父进程一边将它们累积到一个内存缓存中，一边将这些改动追加到现有 AOF 文件的末尾： 这样即使在重写的中途发生停机，现有的 AOF 文件也还是安全的。
4. 当子进程完成重写工作时，它给父进程发送一个信号，父进程在接收到信号之后，将内存缓存中的所有数据追加到新 AOF 文件的末尾。
5. 搞定！现在 Redis 原子地用新文件替换旧文件，之后所有命令都会直接追加到新 AOF 文件的末尾。

# 怎么从 RDB 持久化切换到 AOF 持久化

在 Redis 2.2 或以上版本，可以在不重启的情况下，从 RDB 切换到 AOF ：

1. 为最新的 `dump.rdb` 文件创建一个备份。
2. 将备份放到一个安全的地方。
3. 执行以下两条命令：

>

```
&gt; redis-cli&gt; CONFIG SET appendonly yes &gt; &gt; redis-cli&gt; CONFIG SET save "" &g
```

1. 确保命令执行之后，数据库的键的数量没有改变。
2. 确保写命令会被正确地追加到 AOF 文件的末尾。

步骤 3 执行的第一条命令开启了 AOF 功能： Redis 会阻塞直到初始 AOF 文件创建完成为止， 之后 Redis 会继续处理命令请求， 并开始将写入命令追加到 AOF 文件末尾。

步骤 3 执行的第二条命令用于关闭 RDB 功能。 这一步是可选的， 如果你愿意的话， 也可以同时使用 RDB 和 AOF 这两种持久化功能。

Note

别忘了在 `redis.conf` 中打开 AOF 功能！ 否则的话， 服务器重启之后， 之前通过 `CONFIG SET` 设置的配置就会被遗忘， 程序会按原来的配置来启动服务器。

Note

译注： 原文这里还有介绍 2.0 版本的切换方式， 考虑到 2.0 已经很老旧了， 这里省略了对那部分文档的翻译， 有需要的请参考原文。

# RDB 和 AOF 之间的相互作用

在版本号大于等于 2.4 的 Redis 中， BGSAVE 执行的过程中， 不可以执行 BGREWRITEAOF 。 反过来说， 在 BGREWRITEAOF 执行的过程中， 也不可以执行 BGSAVE 。

这可以防止两个 Redis 后台进程同时对磁盘进行大量的 I/O 操作。

如果 BGSAVE 正在执行， 并且用户显示地调用 BGREWRITEAOF 命令， 那么服务器将向用户回复一个 `OK` 状态， 并告知用户， BGREWRITEAOF 已经被预定执行： 一旦 BGSAVE 执行完毕， BGREWRITEAOF 就会正式开始。

当 Redis 启动时， 如果 RDB 持久化和 AOF 持久化都被打开了， 那么程序会优先使用 AOF 文件来恢复数据集， 因为 AOF 文件所保存的数据通常是最完整的。

# 备份 Redis 数据

在阅读这个小节前， 先将下面这句话铭记于心： 一定要备份你的数据库！

磁盘故障， 节点失效， 诸如此类的问题都可能让你的数据消失不见， 不进行备份是非常危险的。

Redis 对于数据备份是非常友好的， 因为你可以在服务器运行的时候对 RDB 文件进行复制： RDB 文件一旦被创建， 就不会进行任何修改。 当服务器要创建一个新的 RDB 文件时， 它先将文件的内容保存在一个临时文件里面， 当临时文件写入完毕时， 程序才使用 `rename(2)` 原子地用临时文件替换原来的 RDB 文件。

这也就是说， 无论何时， 复制 RDB 文件都是绝对安全的。

以下是我们的建议：

- 创建一个定期任务（cron job）， 每小时将一个 RDB 文件备份到一个文件夹， 并且每天将一个 RDB 文件备份到另一个文件夹。
- 确保快照的备份都带有相应的日期和时间信息， 每次执行定期任务脚本时， 使用 `find` 命令来删除过期的快照： 比如说， 你可以保留最近 48 小时内的每小时快照， 还可以保留最近一两个月的每日快照。
- 至少每天一次， 将 RDB 备份到你的数据中心之外， 或者至少是备份到你运行 Redis 服务器的物理机器之外。

# 容灾备份

Redis 的容灾备份基本上就是对数据进行备份， 并将这些备份传送到多个不同的外部数据中心。

容灾备份可以在 Redis 运行并产生快照的主数据中心发生严重的问题时， 仍然让数据处于安全状态。

因为很多 Redis 用户都是创业者， 他们没有大把大把的钱可以浪费， 所以下面介绍的都是一些实用又便宜的容灾备份方法：

- Amazon S3 ，以及其他类似 S3 的服务，是一个构建灾难备份系统的好地方。 最简单的方法就是将你的每小时或者每日 RDB 备份加密并传送到 S3 。 对数据的加密可以通过 `gpg -c` 命令来完成（对称加密模式）。 记得把你的密码放到几个不同的、安全的地方去（比如你可以把密码复制给你组织里最重要的人物）。 同时使用多个储存服务来保存数据文件，可以提升数据的安全性。
- 传送快照可以使用 SCP 来完成（SSH 的组件）。 以下是简单并且安全的传送方法： 买一个离你的数据中心非常远的 VPS ， 装上 SSH ， 创建一个无口令的 SSH 客户端 key ， 并将这个 key 添加到 VPS 的 authorized_keys 文件中， 这样就可以向这个 VPS 传送快照备份文件了。 为了达到最好的数据安全性，至少要从两个不同的提供商那里各购买一个 VPS 来进行数据容灾备份。

需要注意的是， 这类容灾系统如果没有小心地进行处理的话， 是很容易失效的。

最低限度下， 你应该在文件传送完毕之后， 检查所传送备份文件的体积和原始快照文件的体积是否相同。 如果你使用的是 VPS ， 那么还可以通过比对文件的 SHA1 校验和来确认文件是否传送完整。

另外， 你还需要一个独立的警报系统， 让它在负责传送备份文件的传送器（transfer）失灵时通知你。

# Sentinel

Note

本文档翻译自： http://redis.io/topics/sentinel 。

Redis 的 Sentinel 系统用于管理多个 Redis 服务器（instance）， 该系统执行以下三个任务：

- 监控（**Monitoring**）： Sentinel 会不断地检查你的主服务器和从服务器是否运作正常。
- 提醒（**Notification**）： 当被监控的某个 Redis 服务器出现问题时， Sentinel 可以通过 API 向管理员或者其他应用程序发送通知。
- 自动故障迁移（**Automatic failover**）： 当一个主服务器不能正常工作时， Sentinel 会开始一次自动故障迁移操作， 它会将失效主服务器的其中一个从服务器升级为新的主服务器， 并让失效主服务器的其他从服务器改为复制新的主服务器； 当客户端试图连接失效的主服务器时， 集群也会向客户端返回新主服务器的地址， 使得集群可以使用新主服务器代替失效服务器。

Redis Sentinel 是一个分布式系统， 你可以在一个架构中运行多个 Sentinel 进程（progress）， 这些进程使用流言协议（gossip protocols)来接收关于主服务器是否下线的信息， 并使用投票协议（agreement protocols）来决定是否执行自动故障迁移， 以及选择哪个从服务器作为新的主服务器。

虽然 Redis Sentinel 释出为一个单独的可执行文件 `redis-sentinel` ， 但实际上它只是一个运行在特殊模式下的 Redis 服务器， 你可以在启动一个普通 Redis 服务器时通过给定 `--sentinel` 选项来启动 Redis Sentinel 。

Warning

Redis Sentinel 目前仍在开发中， 这个文档的内容可能随着 Sentinel 实现的修改而变更。

Redis Sentinel 兼容 Redis 2.4.16 或以上版本， 推荐使用 Redis 2.8.0 或以上的版本。

## 获取 Sentinel

目前 Sentinel 系统是 Redis 的 `unstable` 分支的一部分， 你必须到 Redis 项目的 Github 页面 克隆一份 `unstable` 分值， 然后通过编译来获得 Sentinel 系统。

Sentinel 程序可以在编译后的 `src` 文档中发现， 它是一个命名为 `redis-sentinel` 的程序。

你也可以通过下一节介绍的方法， 让 `redis-server` 程序运行在 Sentinel 模式之下。

另外， 一个新版本的 Sentinel 已经包含在了 Redis 2.8.0 版本的释出文件中。

# 启动 Sentinel

对于 `redis-sentinel` 程序， 你可以用以下命令来启动 Sentinel 系统：

```
redis-sentinel /path/to/sentinel.conf
```

对于 `redis-server` 程序， 你可以用以下命令来启动一个运行在 Sentinel 模式下的 Redis 服务器：

```
redis-server /path/to/sentinel.conf --sentinel
```

两种方法都可以启动一个 Sentinel 实例。

启动 Sentinel 实例必须指定相应的配置文件， 系统会使用配置文件来保存 Sentinel 的当前状态， 并在 Sentinel 重启时通过载入配置文件来进行状态还原。

如果启动 Sentinel 时没有指定相应的配置文件， 或者指定的配置文件不可写（not writable）， 那么 Sentinel 会拒绝启动。

# 配置 Sentinel

Redis 源码中包含了一个名为 `sentinel.conf` 的文件， 这个文件是一个带有详细注释的 Sentinel 配置文件示例。

运行一个 Sentinel 所需的最少配置如下所示：

```
sentinel monitor mymaster 127.0.0.1 6379 2
sentinel down-after-milliseconds mymaster 60000
sentinel failover-timeout mymaster 180000
sentinel parallel-syncs mymaster 1

sentinel monitor resque 192.168.1.3 6380 4
sentinel down-after-milliseconds resque 10000
sentinel failover-timeout resque 180000
sentinel parallel-syncs resque 5
```

第一行配置指示 Sentinel 去监视一个名为 `mymaster` 的主服务器， 这个主服务器的 IP 地址为 `127.0.0.1` ， 端口号为 `6379` ， 而将这个主服务器判断为失效至少需要 `2` 个 Sentinel 同意 （只要同意 Sentinel 的数量不达标，自动故障迁移就不会执行）。

不过要注意， 无论你设置要多少个 Sentinel 同意才能判断一个服务器失效， 一个 **Sentinel** 都需要获得系统中多数（**majority**） **Sentinel** 的支持， 才能发起一次自动故障迁移， 并预留一个给定的配置纪元 （configuration Epoch ， 一个配置纪元就是一个新主服务器配置的版本号）。

换句话说， 在只有少数（minority） **Sentinel** 进程正常运作的情况下， **Sentinel** 是不能执行自动故障迁移的。

其他选项的基本格式如下：

```
sentinel <选项的名字> <主服务器的名字> <选项的值>
```

各个选项的功能如下：

- `down-after-milliseconds` 选项指定了 Sentinel 认为服务器已经断线所需的毫秒数。

  如果服务器在给定的毫秒数之内， 没有返回 Sentinel 发送的 PING 命令的回复， 或者返回一个错误， 那么 Sentinel 将这个服务器标记为主观下线（subjectively down， 简称 `SDOWN` ）。

  不过只有一个 Sentinel 将服务器标记为主观下线并不一定会引起服务器的自动故障迁移： 只有在足够数量的 Sentinel 都将一个服务器标记为主观下线之后， 服务器才会被标记为客观下线（objectively down， 简称 `ODOWN` ）， 这时自动故障迁移才会执行。

  将服务器标记为客观下线所需的 Sentinel 数量由对主服务器的配置决定。

- `parallel-syncs` 选项指定了在执行故障转移时， 最多可以有多少个从服务器同时对新的主服务器进行同步， 这个数字越小， 完成故障转移所需的时间就越长。

  如果从服务器被设置为允许使用过期数据集（参见对 `redis.conf` 文件中对 `slave-serve-stale-data` 选项的说明）， 那么你可能不希望所有从服务器都在同一时间向新的主服务器发送同步请求， 因为尽管复制过程的绝大部分步骤都不会阻塞从服务器， 但从服务器在载入主服务器发来的 RDB 文件时， 仍然会造成从服务器在一段时间内不能处理命令请求： 如果全部从服务器一起对新的主服务器进行同步， 那么就可能会造成所有从服务器在短时间内全部不可用的情况出现。

  你可以通过将这个值设为 `1` 来保证每次只有一个从服务器处于不能处理命令请求的状态。

本文档剩余的内容将对 Sentinel 系统的其他选项进行介绍， 示例配置文件 `sentinel.conf` 也对相关的选项进行了完整的注释。

# 主观下线和客观下线

前面说过， Redis 的 Sentinel 中关于下线（down）有两个不同的概念：

- 主观下线（Subjectively Down， 简称 SDOWN）指的是单个 Sentinel 实例对服务器做出的下线判断。
- 客观下线（Objectively Down， 简称 ODOWN）指的是多个 Sentinel 实例在对同一个服务器做出 SDOWN 判断， 并且通过 `SENTINEL is-master-down-by-addr` 命令互相交流之

后， 得出的服务器下线判断。 （一个 Sentinel 可以通过向另一个 Sentinel 发送 `SENTINEL is-master-down-by-addr` 命令来询问对方是否认为给定的服务器已下线。）

如果一个服务器没有在 `master-down-after-milliseconds` 选项所指定的时间内， 对向它发送 PING 命令的 Sentinel 返回一个有效回复（valid reply）， 那么 Sentinel 就会将这个服务器标记为主观下线。

服务器对 PING 命令的有效回复可以是以下三种回复的其中一种：

- 返回 `+PONG` 。
- 返回 `-LOADING` 错误。
- 返回 `-MASTERDOWN` 错误。

如果服务器返回除以上三种回复之外的其他回复， 又或者在指定时间内没有回复 PING 命令， 那么 Sentinel 认为服务器返回的回复无效（non-valid）。

注意， 一个服务器必须在 `master-down-after-milliseconds` 毫秒内， 一直返回无效回复才会被 Sentinel 标记为主观下线。

举个例子， 如果 `master-down-after-milliseconds` 选项的值为 `30000` 毫秒（ `30` 秒）， 那么只要服务器能在每 `29` 秒之内返回至少一次有效回复， 这个服务器就仍然会被认为是处于正常状态的。

从主观下线状态切换到客观下线状态并没有使用严格的法定人数算法（strong quorum algorithm）， 而是使用了流言协议： 如果 Sentinel 在给定的时间范围内， 从其他 Sentinel 那里接收到了足够数量的主服务器下线报告， 那么 Sentinel 就会将主服务器的状态从主观下线改变为客观下线。 如果之后其他 Sentinel 不再报告主服务器已下线， 那么客观下线状态就会被移除。

客观下线条件只适用于主服务器： 对于任何其他类型的 Redis 实例， Sentinel 在将它们判断为下线前不需要进行协商， 所以从服务器或者其他 Sentinel 永远不会达到客观下线条件。

只要一个 Sentinel 发现某个主服务器进入了客观下线状态， 这个 Sentinel 就可能会被其他 Sentinel 推选出， 并对失效的主服务器执行自动故障迁移操作。

# 每个 Sentinel 都需要定期执行的任务

- 每个 Sentinel 以每秒钟一次的频率向它所知的主服务器、从服务器以及其他 Sentinel 实例发送一个 PING 命令。
- 如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 `down-after-milliseconds` 选项所指定的值， 那么这个实例会被 Sentinel 标记为主观下线。 一个有效回复可以是： `+PONG` 、 `-LOADING` 或者 `-MASTERDOWN` 。
- 如果一个主服务器被标记为主观下线， 那么正在监视这个主服务器的所有 Sentinel 要以每秒一次的频率确认主服务器的确进入了主观下线状态。

- 如果一个主服务器被标记为主观下线， 并且有足够数量的 Sentinel （至少要达到配置文件指定的数量）在指定的时间范围内同意这一判断， 那么这个主服务器被标记为客观下线。
- 在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有主服务器和从服务器发送 INFO 命令。 当一个主服务器被 Sentinel 标记为客观下线时， Sentinel 向下线主服务器的所有从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。
- 当没有足够数量的 Sentinel 同意主服务器已经下线， 主服务器的客观下线状态就会被移除。 当主服务器重新向 Sentinel 的 PING 命令返回有效回复时， 主服务器的主管下线状态就会被移除。

# 自动发现 Sentinel 和从服务器

一个 Sentinel 可以与其他多个 Sentinel 进行连接， 各个 Sentinel 之间可以互相检查对方的可用性， 并进行信息交换。

你无须为运行的每个 Sentinel 分别设置其他 Sentinel 的地址， 因为 Sentinel 可以通过发布与订阅功能来自动发现正在监视相同主服务器的其他 Sentinel ， 这一功能是通过向频道 `__sentinel__:hello` 发送信息来实现的。

与此类似， 你也不必手动列出主服务器属下的所有从服务器， 因为 Sentinel 可以通过询问主服务器来获得所有从服务器的信息。

- 每个 Sentinel 会以每两秒一次的频率， 通过发布与订阅功能， 向被它监视的所有主服务器和从服务器的 `__sentinel__:hello` 频道发送一条信息， 信息中包含了 Sentinel 的 IP 地址、端口号和运行 ID （runid）。
- 每个 Sentinel 都订阅了被它监视的所有主服务器和从服务器的 `__sentinel__:hello` 频道， 查找之前未出现过的 sentinel （looking for unknown sentinels）。 当一个 Sentinel 发现一个新的 Sentinel 时， 它会将新的 Sentinel 添加到一个列表中， 这个列表保存了 Sentinel 已知的， 监视同一个主服务器的所有其他 Sentinel 。
- Sentinel 发送的信息中还包括完整的主服务器当前配置（configuration）。 如果一个 Sentinel 包含的主服务器配置比另一个 Sentinel 发送的配置要旧， 那么这个 Sentinel 会立即升级到新配置上。
- 在将一个新 Sentinel 添加到监视主服务器的列表上面之前， Sentinel 会先检查列表中是否已经包含了和要添加的 Sentinel 拥有相同运行 ID 或者相同地址（包括 IP 地址和端口号）的 Sentinel ， 如果是的话， Sentinel 会先移除列表中已有的那些拥有相同运行 ID 或者相同地址的 Sentinel ， 然后再添加新 Sentinel 。

# Sentinel API

在默认情况下， Sentinel 使用 TCP 端口 `26379` （普通 Redis 服务器使用的是 `6379` ）。

Sentinel 接受 Redis 协议格式的命令请求， 所以你可以使用 `redis-cli` 或者任何其他 Redis 客户端来与 Sentinel 进行通讯。

有两种方式可以和 Sentinel 进行通讯：

- 第一种方法是通过直接发送命令来查询被监视 Redis 服务器的当前状态， 以及 Sentinel 所知道的关于其他 Sentinel 的信息， 诸如此类。
- 另一种方法是使用发布与订阅功能， 通过接收 Sentinel 发送的通知： 当执行故障转移操作， 或者某个被监视的服务器被判断为主观下线或者客观下线时， Sentinel 就会发送相应的信息。

## Sentinel 命令

以下列出的是 Sentinel 接受的命令：

- PING ：返回 `PONG` 。
- `SENTINEL masters` ：列出所有被监视的主服务器，以及这些主服务器的当前状态。
- `SENTINEL slaves <master name>` ：列出给定主服务器的所有从服务器，以及这些从服务器的当前状态。
- `SENTINEL get-master-addr-by-name <master name>` ： 返回给定名字的主服务器的 IP 地址和端口号。 如果这个主服务器正在执行故障转移操作， 或者针对这个主服务器的故障转移操作已经完成， 那么这个命令返回新的主服务器的 IP 地址和端口号。
- `SENTINEL reset <pattern>` ： 重置所有名字和给定模式 `pattern` 相匹配的主服务器。 `pattern` 参数是一个 Glob 风格的模式。 重置操作清除主服务器目前的所有状态， 包括正在执行中的故障转移， 并移除目前已经发现和关联的， 主服务器的所有从服务器和 Sentinel 。
- `SENTINEL failover <master name>` ： 当主服务器失效时， 在不询问其他 Sentinel 意见的情况下， 强制开始一次自动故障迁移 （不过发起故障转移的 Sentinel 会向其他 Sentinel 发送一个新的配置，其他 Sentinel 会根据这个配置进行相应的更新）。

## 发布与订阅信息

客户端可以将 Sentinel 看作是一个只提供了订阅功能的 Redis 服务器： 你不可以使用 PUBLISH 命令向这个服务器发送信息， 但你可以用 SUBSCRIBE 命令或者 PSUBSCRIBE 命令， 通过订阅给定的频道来获取相应的事件提醒。

一个频道能够接收和这个频道的名字相同的事件。 比如说， 名为 `+sdown` 的频道就可以接收所有实例进入主观下线（SDOWN）状态的事件。

通过执行 `PSUBSCRIBE *` 命令可以接收所有事件信息。

以下列出的是客户端可以通过订阅来获得的频道和信息的格式： 第一个英文单词是频道/事件的名字， 其余的是数据的格式。

注意， 当格式中包含 `instance details` 字样时， 表示频道所返回的信息中包含了以下用于识别目标实例的内容：

```
<instance-type> <name> <ip> <port> @ <master-name> <master-ip> <master-port>
```

`@` 字符之后的内容用于指定主服务器， 这些内容是可选的， 它们仅在 `@` 字符之前的内容指定的实例不是主服务器时使用。

- `+reset-master <instance details>` ：主服务器已被重置。
- `+slave <instance details>` ：一个新的从服务器已经被 Sentinel 识别并关联。
- `+failover-state-reconf-slaves <instance details>` ：故障转移状态切换到了 `reconf-slaves` 状态。
- `+failover-detected <instance details>` ：另一个 Sentinel 开始了一次故障转移操作，或者一个从服务器转换成了主服务器。
- `+slave-reconf-sent <instance details>` ：领头（leader）的 Sentinel 向实例发送了 SLAVEOF 命令，为实例设置新的主服务器。
- `+slave-reconf-inprog <instance details>` ：实例正在将自己设置为指定主服务器的从服务器，但相应的同步过程仍未完成。
- `+slave-reconf-done <instance details>` ：从服务器已经成功完成对新主服务器的同步。
- `-dup-sentinel <instance details>` ：对给定主服务器进行监视的一个或多个 Sentinel 已经因为重复出现而被移除 —— 当 Sentinel 实例重启的时候，就会出现这种情况。
- `+sentinel <instance details>` ：一个监视给定主服务器的新 Sentinel 已经被识别并添加。
- `+sdown <instance details>` ：给定的实例现在处于主观下线状态。
- `-sdown <instance details>` ：给定的实例已经不再处于主观下线状态。
- `+odown <instance details>` ：给定的实例现在处于客观下线状态。
- `-odown <instance details>` ：给定的实例已经不再处于客观下线状态。
- `+new-epoch <instance details>` ：当前的纪元（epoch）已经被更新。
- `+try-failover <instance details>` ：一个新的故障迁移操作正在执行中，等待被大多数 Sentinel 选中（waiting to be elected by the majority）。
- `+elected-leader <instance details>` ：赢得指定纪元的选举，可以进行故障迁移操作了。
- `+failover-state-select-slave <instance details>` ：故障转移操作现在处于 `select-slave` 状态 —— Sentinel 正在寻找可以升级为主服务器的从服务器。
- `no-good-slave <instance details>` ：Sentinel 操作未能找到适合进行升级的从服务器。Sentinel 会在一段时间之后再次尝试寻找合适的从服务器来进行升级，又或者直接放弃执行故障转移操作。
- `selected-slave <instance details>` ：Sentinel 顺利找到适合进行升级的从服务器。

- `failover-state-send-slaveof-noone <instance details>` ：Sentinel 正在将指定的从服务器升级为主服务器，等待升级功能完成。
- `failover-end-for-timeout <instance details>` ：故障转移因为超时而中止，不过最终所有从服务器都会开始复制新的主服务器（slaves will eventually be configured to replicate with the new master anyway）。
- `failover-end <instance details>` ：故障转移操作顺利完成。所有从服务器都开始复制新的主服务器了。
- `+switch-master <master name> <oldip> <oldport> <newip> <new` ：配置变更，主服务器的 IP 和地址已经改变。 这是绝大多数外部用户都关心的信息。
- `+tilt` ：进入 tilt 模式。
- `-tilt` ：退出 tilt 模式。

# 故障转移

一次故障转移操作由以下步骤组成：

- 发现主服务器已经进入客观下线状态。
- 对我们的当前纪元进行自增（详情请参考 Raft leader election ）， 并尝试在这个纪元中当选。
- 如果当选失败， 那么在设定的故障迁移超时时间的两倍之后， 重新尝试当选。 如果当选成功， 那么执行以下步骤。
- 选出一个从服务器，并将它升级为主服务器。
- 向被选中的从服务器发送 `SLAVEOF NO ONE` 命令，让它转变为主服务器。
- 通过发布与订阅功能， 将更新后的配置传播给所有其他 Sentinel ， 其他 Sentinel 对它们自己的配置进行更新。
- 向已下线主服务器的从服务器发送 SLAVEOF 命令， 让它们去复制新的主服务器。
- 当所有从服务器都已经开始复制新的主服务器时， 领头 Sentinel 终止这次故障迁移操作。

Note

每当一个 Redis 实例被重新配置（reconfigured） —— 无论是被设置成主服务器、从服务器、又或者被设置成其他主服务器的从服务器 —— Sentinel 都会向被重新配置的实例发送一个 `CONFIG REWRITE` 命令， 从而确保这些配置会持久化在硬盘里。

Sentinel 使用以下规则来选择新的主服务器：

- 在失效主服务器属下的从服务器当中， 那些被标记为主观下线、已断线、或者最后一次回复 PING 命令的时间大于五秒钟的从服务器都会被淘汰。
- 在失效主服务器属下的从服务器当中， 那些与失效主服务器连接断开的时长超过 `down-after` 选项指定的时长十倍的从服务器都会被淘汰。
- 在经历了以上两轮淘汰之后剩下来的从服务器中， 我们选出复制偏移量（replication

offset）最大的那个从服务器作为新的主服务器；如果复制偏移量不可用，或者从服务器的复制偏移量相同，那么带有最小运行 ID 的那个从服务器成为新的主服务器。

## Sentinel 自动故障迁移的一致性特质

Sentinel 自动故障迁移使用 Raft 算法来选举领头（leader） Sentinel ，从而确保在一个给定的纪元（epoch）里，只有一个领头产生。

这表示在同一个纪元中，不会有两个 Sentinel 同时被选中为领头，并且各个 Sentinel 在同一个纪元中只会对一个领头进行投票。

更高的配置纪元总是优于较低的纪元，因此每个 Sentinel 都会主动使用更新的纪元来代替自己的配置。

简单来说，我们可以将 Sentinel 配置看作是一个带有版本号的状态。一个状态会以最后写入者胜出（last-write-wins）的方式（也即是，最新的配置总是胜出）传播至所有其他 Sentinel 。

举个例子，当出现网络分割（network partitions）时，一个 Sentinel 可能会包含了较旧的配置，而当这个 Sentinel 接到其他 Sentinel 发来的版本更新的配置时，Sentinel 就会对自己的配置进行更新。

如果要在网络分割出现的情况下仍然保持一致性，那么应该使用 `min-slaves-to-write` 选项，让主服务器在连接的从实例少于给定数量时停止执行写操作，与此同时，应该在每个运行 Redis 主服务器或从服务器的机器上运行 Redis Sentinel 进程。

## Sentinel 状态的持久化

Sentinel 的状态会被持久化在 Sentinel 配置文件里面。

每当 Sentinel 接收到一个新的配置，或者当领头 Sentinel 为主服务器创建一个新的配置时，这个配置会与配置纪元一起被保存到磁盘里面。

这意味着停止和重启 Sentinel 进程都是安全的。

## Sentinel 在非故障迁移的情况下对实例进行重新配置

即使没有自动故障迁移操作在进行，Sentinel 总会尝试将当前的配置设置到被监视的实例上面。特别是：

- 根据当前的配置，如果一个从服务器被宣告为主服务器，那么它会代替原有的主服务器，成为新的主服务器，并且成为原有主服务器的所有从服务器的复制对象。
- 那些连接了错误主服务器的从服务器会被重新配置，使得这些从服务器会去复制正确的主服务器。

不过， 在以上这些条件满足之后， Sentinel 在对实例进行重新配置之前仍然会等待一段足够长的时间， 确保可以接收到其他 Sentinel 发来的配置更新， 从而避免自身因为保存了过期的配置而对实例进行了不必要的重新配置。

# TILT 模式

Redis Sentinel 严重依赖计算机的时间功能： 比如说， 为了判断一个实例是否可用， Sentinel 会记录这个实例最后一次相应 PING 命令的时间， 并将这个时间和当前时间进行对比， 从而知道这个实例有多长时间没有和 Sentinel 进行任何成功通讯。

不过， 一旦计算机的时间功能出现故障， 或者计算机非常忙碌， 又或者进程因为某些原因而被阻塞时， Sentinel 可能也会跟着出现故障。

TILT 模式是一种特殊的保护模式： 当 Sentinel 发现系统有些不对劲时， Sentinel 就会进入 TILT 模式。

因为 Sentinel 的时间中断器默认每秒执行 10 次， 所以我们预期时间中断器的两次执行之间的间隔为 100 毫秒左右。 Sentinel 的做法是， 记录上一次时间中断器执行时的时间， 并将它和这一次时间中断器执行的时间进行对比：

- 如果两次调用时间之间的差距为负值， 或者非常大（超过 2 秒钟）， 那么 Sentinel 进入 TILT 模式。
- 如果 Sentinel 已经进入 TILT 模式， 那么 Sentinel 延迟退出 TILT 模式的时间。

当 Sentinel 进入 TILT 模式时， 它仍然会继续监视所有目标， 但是：

- 它不再执行任何操作，比如故障转移。
- 当有实例向这个 Sentinel 发送 `SENTINEL is-master-down-by-addr` 命令时， Sentinel 返回负值： 因为这个 Sentinel 所进行的下线判断已经不再准确。

如果 TILT 可以正常维持 30 秒钟， 那么 Sentinel 退出 TILT 模式。

# 处理 `-BUSY` 状态

Warning

该功能尚未实现

当 Lua 脚本的运行时间超过指定时限时， Redis 就会返回 `-BUSY` 错误。

当出现这种情况时， Sentinel 在尝试执行故障转移操作之前， 会先向服务器发送一个 SCRIPT KILL 命令， 如果服务器正在执行的是一个只读脚本的话， 那么这个脚本就会被杀死， 服务器就会回到正常状态。

# Sentinel 的客户端实现

关于 Sentinel 客户端的实现信息可以参考 Sentinel 客户端指引手册 。

# 集群教程

Note

本文档翻译自 http://redis.io/topics/cluster-tutorial 。

本文档是 Redis 集群的入门教程， 从用户的角度介绍了设置、测试和操作集群的方法。

本教程不包含晦涩难懂的分布式概念， 也没有像 Redis 集群规范 那样包含 Redis 集群的实现细节， 如果你打算深入地学习 Redis 集群的部署方法， 那么推荐你在阅读完这个教程之后， 再去看一看集群规范。

**Redis** 集群目前仍处于 **Alpha** 测试版本， 如果在使用过程中发现任何问题， 请到 Redis 的邮件列表 发贴， 或者到 Redis 的 Github 页面 报告错误。

# 集群简介

Redis 集群是一个可以在多个 **Redis** 节点之间进行数据共享的设施（installation）。

Redis 集群不支持那些需要同时处理多个键的 Redis 命令， 因为执行这些命令需要在多个 Redis 节点之间移动数据， 并且在高负载的情况下， 这些命令将降低 Redis 集群的性能， 并导致不可预测的行为。

Redis 集群通过分区（**partition**）来提供一定程度的可用性（availability）： 即使集群中有一部分节点失效或者无法进行通讯， 集群也可以继续处理命令请求。

Redis 集群提供了以下两个好处：

- 将数据自动切分（split）到多个节点的能力。
- 当集群中的一部分节点失效或者无法进行通讯时， 仍然可以继续处理命令请求的能力。

# Redis 集群数据共享

Redis 集群使用数据分片（sharding）而非一致性哈希（consistency hashing）来实现： 一个 Redis 集群包含 `16384` 个哈希槽（hash slot）， 数据库中的每个键都属于这 `16384` 个哈希槽的其中一个， 集群使用公式 `CRC16(key) % 16384` 来计算键 `key` 属于哪个槽， 其中 `CRC16(key)` 语句用于计算键 `key` 的 CRC16 校验和 。

集群中的每个节点负责处理一部分哈希槽。 举个例子， 一个集群可以有三个哈希槽， 其中：

- 节点 A 负责处理 `0` 号至 `5500` 号哈希槽。
- 节点 B 负责处理 `5501` 号至 `11000` 号哈希槽。

- 节点 C 负责处理 `11001` 号至 `16384` 号哈希槽。

这种将哈希槽分布到不同节点的做法使得用户可以很容易地向集群中添加或者删除节点。 比如说：

- 如果用户将新节点 D 添加到集群中， 那么集群只需要将节点 A 、B 、 C 中的某些槽移动到节点 D 就可以了。
- 与此类似， 如果用户要从集群中移除节点 A ， 那么集群只需要将节点 A 中的所有哈希槽移动到节点 B 和节点 C ， 然后再移除空白（不包含任何哈希槽）的节点 A 就可以了。

因为将一个哈希槽从一个节点移动到另一个节点不会造成节点阻塞， 所以无论是添加新节点还是移除已存在节点， 又或者改变某个节点包含的哈希槽数量， 都不会造成集群下线。

# Redis 集群中的主从复制

为了使得集群在一部分节点下线或者无法与集群的大多数（majority）节点进行通讯的情况下， 仍然可以正常运作， Redis 集群对节点使用了主从复制功能： 集群中的每个节点都有 `1` 个至 `N` 个复制品（replica）， 其中一个复制品为主节点（master）， 而其余的 `N-1` 个复制品为从节点（slave）。

在之前列举的节点 A 、B 、C 的例子中， 如果节点 B 下线了， 那么集群将无法正常运行， 因为集群找不到节点来处理 `5501` 号至 `11000` 号的哈希槽。

另一方面， 假如在创建集群的时候（或者至少在节点 B 下线之前）， 我们为主节点 B 添加了从节点 B1 ， 那么当主节点 B 下线的时候， 集群就会将 B1 设置为新的主节点， 并让它代替下线的主节点 B ， 继续处理 `5501` 号至 `11000` 号的哈希槽， 这样集群就不会因为主节点 B 的下线而无法正常运作了。

不过如果节点 B 和 B1 都下线的话， Redis 集群还是会停止运作。

# Redis 集群的一致性保证（guarantee）

Redis 集群不保证数据的强一致性（strong consistency）： 在特定条件下， Redis 集群可能会丢失已经被执行过的写命令。

使用异步复制（asynchronous replication）是 Redis 集群可能会丢失写命令的其中一个原因。 考虑以下这个写命令的例子：

- 客户端向主节点 B 发送一条写命令。
- 主节点 B 执行写命令，并向客户端返回命令回复。
- 主节点 B 将刚刚执行的写命令复制给它的从节点 B1 、 B2 和 B3 。

如你所见， 主节点对命令的复制工作发生在返回命令回复之后， 因为如果每次处理命令请求都需要等待复制操作完成的话， 那么主节点处理命令请求的速度将极大地降低 —— 我们必须在性能和一致性之间做出权衡。

Note

如果真的有必要的话， Redis 集群可能会在将来提供同步地（synchronou）执行写命令的方法。

Redis 集群另外一种可能会丢失命令的情况是， 集群出现网络分裂（network partition）， 并且一个客户端与至少包括一个主节点在内的少数（minority）实例被孤立。

举个例子， 假设集群包含 A 、 B 、 C 、 A1 、 B1 、 C1 六个节点， 其中 A 、B 、C 为主节点， 而 A1 、B1 、C1 分别为三个主节点的从节点， 另外还有一个客户端 Z1 。

假设集群中发生网络分裂， 那么集群可能会分裂为两方， 大多数（majority）的一方包含节点 A 、C 、A1 、B1 和 C1 ， 而少数（minority）的一方则包含节点 B 和客户端 Z1 。

在网络分裂期间， 主节点 B 仍然会接受 Z1 发送的写命令：

- 如果网络分裂出现的时间很短， 那么集群会继续正常运行；
- 但是， 如果网络分裂出现的时间足够长， 使得大多数一方将从节点 B1 设置为新的主节点， 并使用 B1 来代替原来的主节点 B ， 那么 Z1 发送给主节点 B 的写命令将丢失。

注意， 在网络分裂出现期间， 客户端 Z1 可以向主节点 B 发送写命令的最大时间是有限制的， 这一时间限制称为节点超时时间（node timeout）， 是 Redis 集群的一个重要的配置选项：

- 对于大多数一方来说， 如果一个主节点未能在节点超时时间所设定的时限内重新联系上集群， 那么集群会将这个主节点视为下线， 并使用从节点来代替这个主节点继续工作。
- 对于少数一方， 如果一个主节点未能在节点超时时间所设定的时限内重新联系上集群， 那么它将停止处理写命令， 并向客户端报告错误。

# 创建并使用 Redis 集群

Redis 集群由多个运行在集群模式（cluster mode）下的 Redis 实例组成， 实例的集群模式需要通过配置来开启， 开启集群模式的实例将可以使用集群特有的功能和命令。

以下是一个包含了最少选项的集群配置文件示例：

```
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

文件中的 `cluster-enabled` 选项用于开实例的集群模式， 而 `cluster-conf-file` 选项则设定了保存节点配置文件的路径， 默认值为 `nodes.conf` 。

节点配置文件无须人为修改， 它由 Redis 集群在启动时创建， 并在有需要时自动进行更新。

要让集群正常运作至少需要三个主节点， 不过在刚开始试用集群功能时， 强烈建议使用六个节点： 其中三个为主节点， 而其余三个则是各个主节点的从节点。

首先， 让我们进入一个新目录， 并创建六个以端口号为名字的子目录， 稍后我们在将每个目录中运行一个 Redis 实例：

```
mkdir cluster-test
cd cluster-test
mkdir 7000 7001 7002 7003 7004 7005
```

在文件夹 `7000` 至 `7005` 中， 各创建一个 `redis.conf` 文件， 文件的内容可以使用上面的示例配置文件， 但记得将配置中的端口号从 `7000` 改为与文件夹名字相同的号码。

现在， 从 Redis Github 页面 的 `unstable` 分支中取出最新的 Redis 源码， 编译出可执行文件 `redis-server` ， 并将文件复制到 `cluster-test` 文件夹， 然后使用类似以下命令， 在每个标签页中打开一个实例：

```
cd 7000
../redis-server ./redis.conf
```

实例打印的日志显示， 因为 `nodes.conf` 文件不存在， 所以每个节点都为它自身指定了一个新的 ID ：

```
[82462] 26 Nov 11:56:55.329 * No cluster configuration found, I'm 97a3a64667477371c447932
```

实例会一直使用同一个 ID ， 从而在集群中保持一个独一无二（unique）的名字。

每个节点都使用 ID 而不是 IP 或者端口号来记录其他节点， 因为 IP 地址和端口号都可能会改变， 而这个独一无二的标识符（identifier）则会在节点的整个生命周期中一直保持不变。

我们将这个标识符称为节点 **ID**。

# 创建集群

现在我们已经有了六个正在运行中的 Redis 实例， 接下来我们需要使用这些实例来创建集群， 并为每个节点编写配置文件。

通过使用 Redis 集群命令行工具 `redis-trib` ， 编写节点配置文件的工作可以非常容易地完成： `redis-trib` 位于 Redis 源码的 `src` 文件夹中， 它是一个 Ruby 程序， 这个程序通过向实例发送特殊命令来完成创建新集群， 检查集群， 或者对集群进行重新分片（reshared）等工作。

我们需要执行以下命令来创建集群：

```
./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005
```

命令的意义如下：

- 给定 `redis-trib.rb` 程序的命令是 `create` ， 这表示我们希望创建一个新的集群。
- 选项 `--replicas 1` 表示我们希望为集群中的每个主节点创建一个从节点。
- 之后跟着的其他参数则是实例的地址列表， 我们希望程序使用这些地址所指示的实例来创建新集群。

简单来说， 以上命令的意思就是让 `redis-trib` 程序创建一个包含三个主节点和三个从节点的集群。

接着， `redis-trib` 会打印出一份预想中的配置给你看， 如果你觉得没问题的话， 就可以输入 `yes` ， `redis-trib` 就会将这份配置应用到集群当中：

```
>>> Creating cluster
Connecting to node 127.0.0.1:7000: OK
Connecting to node 127.0.0.1:7001: OK
Connecting to node 127.0.0.1:7002: OK
Connecting to node 127.0.0.1:7003: OK
Connecting to node 127.0.0.1:7004: OK
Connecting to node 127.0.0.1:7005: OK
>>> Performing hash slots allocation on 6 nodes...
Using 3 masters:
127.0.0.1:7000
127.0.0.1:7001
127.0.0.1:7002
127.0.0.1:7000 replica #1 is 127.0.0.1:7003
127.0.0.1:7001 replica #1 is 127.0.0.1:7004
127.0.0.1:7002 replica #1 is 127.0.0.1:7005
M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000
slots:0-5460 (5461 slots) master
M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001
slots:5461-10921 (5461 slots) master
M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002
slots:10922-16383 (5462 slots) master
S: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003
S: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004
S: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005
Can I set the above configuration? (type 'yes' to accept): yes
```

输入 `yes` 并按下回车确认之后， 集群就会将配置应用到各个节点， 并连接起（join）各个节点 —— 也即是， 让各个节点开始互相通讯：

```
>>> Nodes configuration updated
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join...
>>> Performing Cluster Check (using node 127.0.0.1:7000)
M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000
slots:0-5460 (5461 slots) master
M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001
slots:5461-10921 (5461 slots) master
M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002
slots:10922-16383 (5462 slots) master
M: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003
slots: (0 slots) master
M: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004
slots: (0 slots) master
M: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005
slots: (0 slots) master
[OK] All nodes agree about slots configuration.
```

如果一切正常的话， `redis-trib` 将输出以下信息：

```
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

这表示集群中的 `16384` 个槽都有至少一个主节点在处理， 集群运作正常。

# 集群的客户端

Redis 集群现阶段的一个问题是客户端实现很少。 以下是一些我知道的实现：

- `redis-rb-cluster` 是我（@antirez）编写的 Ruby 实现， 用于作为其他实现的参考。 该实现是对 `redis-rb` 的一个简单包装， 高效地实现了与集群进行通讯所需的最少语义（semantic）。
- `redis-py-cluster` 看上去是 `redis-rb-cluster` 的一个 Python 版本， 这个项目有一段时间没有更新了（最后一次提交是在六个月之前）， 不过可以将这个项目用作学习集群的起点。
- 流行的 Predis 曾经对早期的 Redis 集群有过一定的支持， 但我不确定它对集群的支持是否完整， 也不清楚它是否和最新版本的 Redis 集群兼容 （因为新版的 Redis 集群将槽的数量从 4k 改为 16k 了）。
- Redis `unstable` 分支中的 `redis-cli` 程序实现了非常基本的集群支持， 可以使用命令 `redis-cli -c` 来启动。

测试 Redis 集群比较简单的办法就是使用 `redis-rb-cluster` 或者 `redis-cli` ， 接下来我们将使用 `redis-cli` 为例来进行演示：

```
$ redis-cli -c -p 7000
redis 127.0.0.1:7000> set foo bar
-> Redirected to slot [12182] located at 127.0.0.1:7002
OK

redis 127.0.0.1:7002> set hello world
-> Redirected to slot [866] located at 127.0.0.1:7000
OK

redis 127.0.0.1:7000> get foo
-> Redirected to slot [12182] located at 127.0.0.1:7002
"bar"

redis 127.0.0.1:7000> get hello
-> Redirected to slot [866] located at 127.0.0.1:7000
"world"
```

`redis-cli` 对集群的支持是非常基本的， 所以它总是依靠 Redis 集群节点来将它转向（redirect）至正确的节点。

一个真正的（serious）集群客户端应该做得比这更好： 它应该用缓存记录起哈希槽与节点地址之间的映射（map）， 从而直接将命令发送到正确的节点上面。

这种映射只会在集群的配置出现某些修改时变化， 比如说， 在一次故障转移（failover）之后， 或者系统管理员通过添加节点或移除节点来修改了集群的布局（layout）之后， 诸如此类。

# 使用 `redis-rb-cluster` 编写一个示例应用

在展示如何使用集群进行故障转移、重新分片等操作之前， 我们需要创建一个示例应用， 了解一些与 Redis 集群客户端进行交互的基本方法。

在运行示例应用的过程中， 我们会尝试让节点进入失效状态， 又或者开始一次重新分片， 以此来观察 Redis 集群在真实世界运行时的表现， 并且为了让这个示例尽可能地有用， 我们会让这个应用向集群进行写操作。

本节将通过两个示例应用来展示 `redis-rb-cluster` 的基本用法， 以下是本节的第一个示例应用， 它是一个名为 example.rb 的文件， 包含在redis-rb-cluster 项目里面：

|

|

```
require './cluster'

startup_nodes = [
    {:host =&gt; "127.0.0.1", :port =&gt; 7000},
    {:host =&gt; "127.0.0.1", :port =&gt; 7001}
]
rc = RedisCluster.new(startup_nodes,32,:timeout =&gt; 0.1)

last = false

while not last
    begin
        last = rc.get("__last__")
        last = 0 if !last
    rescue =&gt; e
        puts "error #{e.to_s}"
        sleep 1
    end
end

((last.to_i+1)..1000000000).each{&#124;x&#124;
    begin
        rc.set("foo#{x}",x)
        puts rc.get("foo#{x}")
        rc.set("__last__",x)
    rescue =&gt; e
        puts "error #{e.to_s}"
    end
    sleep 0.1
}
```

|

这个应用所做的工作非常简单： 它不断地以 `foo<number>` 为键， `number` 为值， 使用 SET 命令向数据库设置键值对。

如果我们执行这个应用的话， 应用将按顺序执行以下命令：

- `SET foo0 0`
- `SET foo1 1`
- `SET foo2 2`
- 诸如此类。。。

代码中的每个集群操作都使用一个 `begin` 和 `rescue` 代码块（block）包裹着， 因为我们希望在代码出错时， 将错误打印到终端上面， 而不希望应用因为异常（exception）而退出。

代码的第七行是代码中第一个有趣的地方， 它创建了一个 Redis 集群对象， 其中创建对象所使用的参数及其意义如下：

- 第一个参数是记录了启动节点的 `startup_nodes` 列表， 列表中包含了两个集群节点的地址。
- 第二个参数指定了对于集群中的各个不同的节点， Redis 集群对象可以获得（take）的最大连接数 （maximum number of connections this object is allowed to take）。
- 第三个参数 `timeout` 指定了一个命令在执行多久之后， 才会被看作是执行失败。

记住， 启动列表中并不需要包含所有集群节点的地址， 但这些地址中至少要有一个是有效的（reachable）： 一旦 `redis-rb-cluster` 成功连接上集群中的某个节点时， 集群节点列表就会被自动更新， 任何真正的（serious）的集群客户端都应该这样做。

现在， 程序创建的 Redis 集群对象实例被保存到 `rc` 变量里面， 我们可以将这个对象当作普通 Redis 对象实例来使用。

在十一至十九行， 我们先尝试阅读计数器中的值， 如果计数器不存在的话， 我们才将计数器初始化为 `0` ： 通过将计数值保存到 Redis 的计数器里面， 我们可以在示例重启之后， 仍然继续之前的执行过程， 而不必每次重启之后都从 `foo0` 开始重新设置键值对。

为了让程序在集群下线的情况下， 仍然不断地尝试读取计数器的值， 我们将读取操作包含在了一个 `while` 循环里面， 一般的应用程序并不需要如此小心。

二十一至三十行是程序的主循环， 这个循环负责设置键值对， 并在设置出错时打印错误信息。

程序在主循环的末尾添加了一个 `sleep` 调用， 让写操作的执行速度变慢， 帮助执行示例的人更容易看清程序的输出。

执行 `example.rb` 程序将产生以下输出：

```
ruby ./example.rb
1
2
3
4
5
6
7
8
9
...
```

这个程序并不是十分有趣， 稍后我们就会看到一个更有趣的集群应用示例， 不过在此之前， 让我们先使用这个示例来演示集群的重新分片操作。

# 对集群进行重新分片

现在， 让我们来试试对集群进行重新分片操作。

在执行重新分片的过程中， 请让你的 `example.rb` 程序处于运行状态， 这样你就会看到， 重新分片并不会对正在运行的集群程序产生任何影响， 你也可以考虑将 `example.rb` 中的 `sleep` 调用删掉， 从而让重新分片操作在近乎真实的写负载下执行。

重新分片操作基本上就是将某些节点上的哈希槽移动到另外一些节点上面， 和创建集群一样， 重新分片也可以使用 `redis-trib` 程序来执行。

执行以下命令可以开始一次重新分片操作：

```
$ ./redis-trib.rb reshard 127.0.0.1:7000
```

你只需要指定集群中其中一个节点的地址， `redis-trib` 就会自动找到集群中的其他节点。

目前 `redis-trib` 只能在管理员的协助下完成重新分片的工作， 要让 `redis-trib` 自动将哈希槽从一个节点移动到另一个节点， 目前来说还做不到 （不过实现这个功能并不难）。

执行 `redis-trib` 的第一步就是设定你打算移动的哈希槽的数量：

```
$ ./redis-trib.rb reshard 127.0.0.1:7000
Connecting to node 127.0.0.1:7000: OK
Connecting to node 127.0.0.1:7002: OK
Connecting to node 127.0.0.1:7005: OK
Connecting to node 127.0.0.1:7001: OK
Connecting to node 127.0.0.1:7003: OK
Connecting to node 127.0.0.1:7004: OK
>>> Performing Cluster Check (using node 127.0.0.1:7000)
M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000
slots:0-5460 (5461 slots) master
M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002
slots:10922-16383 (5462 slots) master
S: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005
slots: (0 slots) slave
M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001
slots:5461-10921 (5461 slots) master
S: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003
slots: (0 slots) slave
S: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004
slots: (0 slots) slave
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
How many slots do you want to move (from 1 to 16384)? 1000
```

我们将打算移动的槽数量设置为 `1000` 个， 如果 `example.rb` 程序一直运行着的话， 现在 `1000` 个槽里面应该有不少键了。

除了移动的哈希槽数量之外， `redis-trib` 还需要知道重新分片的目标（target node）， 也即是， 负责接收这 `1000` 个哈希槽的节点。

指定目标需要使用节点的 ID ， 而不是 IP 地址和端口。 比如说， 我们打算使用集群的第一个主节点来作为目标， 它的 IP 地址和端口是 `127.0.0.1:7000` ， 而节点 ID 则是 `9991306f0e50640a5684f1958fd754b38fa034c9` ， 那么我们应该向 `redis-trib` 提供节点的 ID ：

```
$ ./redis-trib.rb reshard 127.0.0.1:7000
...
What is the receiving node ID? 9991306f0e50640a5684f1958fd754b38fa034c9
```

Note

`redis-trib` 会打印出集群中所有节点的 ID ， 并且我们也可以通过执行以下命令来获得节点的运行 ID ：

```
$ ./redis-cli -p 7000 cluster nodes | grep myself
9991306f0e50640a5684f1958fd754b38fa034c9 :0 myself,master - 0 0 0 connected 0-5460
```

接着， `redis-trib` 会向你询问重新分片的源节点（source node）， 也即是， 要从哪个节点中取出 `1000` 个哈希槽， 并将这些槽移动到目标节点上面。

如果我们不打算从特定的节点上取出指定数量的哈希槽， 那么可以向 `redis-trib` 输入 `all` ， 这样的话， 集群中的所有主节点都会成为源节点， `redis-trib` 将从各个源节点中各取出一部分哈希槽， 凑够 `1000` 个， 然后移动到目标节点上面：

```
$ ./redis-trib.rb reshard 127.0.0.1:7000
...
Please enter all the source node IDs.
Type 'all' to use all the nodes as source nodes for the hash slots.
Type 'done' once you entered all the source nodes IDs.
Source node #1:all
```

输入 `all` 并按下回车之后， `redis-trib` 将打印出哈希槽的移动计划， 如果你觉得没问题的话， 就可以输入 `yes` 并再次按下回车：

```
$ ./redis-trib.rb reshard 127.0.0.1:7000
...
Moving slot 11421 from 393c6df5eb4b4cec323f0e4ca961c8b256e3460a
Moving slot 11422 from 393c6df5eb4b4cec323f0e4ca961c8b256e3460a
Moving slot 5461 from e68e52cee0550f558b03b342f2f0354d2b8a083b
Moving slot 5469 from e68e52cee0550f558b03b342f2f0354d2b8a083b
...
Moving slot 5959 from e68e52cee0550f558b03b342f2f0354d2b8a083b
Do you want to proceed with the proposed reshard plan (yes/no)? yes
```

输入 `yes` 并使用按下回车之后， `redis-trib` 就会正式开始执行重新分片操作， 将指定的哈希槽从源节点一个个地移动到目标节点上面：

```
$ ./redis-trib.rb reshard 127.0.0.1:7000
...
Moving slot 5934 from 127.0.0.1:7001 to 127.0.0.1:7000:
Moving slot 5935 from 127.0.0.1:7001 to 127.0.0.1:7000:
Moving slot 5936 from 127.0.0.1:7001 to 127.0.0.1:7000:
Moving slot 5937 from 127.0.0.1:7001 to 127.0.0.1:7000:
...
Moving slot 5959 from 127.0.0.1:7001 to 127.0.0.1:7000:
```

在重新分片的过程中， `example.rb` 应该可以继续正常运行， 不会出现任何问题。

在重新分片操作执行完毕之后， 可以使用以下命令来检查集群是否正常：

```
$ ./redis-trib.rb check 127.0.0.1:7000
Connecting to node 127.0.0.1:7000: OK
Connecting to node 127.0.0.1:7002: OK
Connecting to node 127.0.0.1:7005: OK
Connecting to node 127.0.0.1:7001: OK
Connecting to node 127.0.0.1:7003: OK
Connecting to node 127.0.0.1:7004: OK
>>> Performing Cluster Check (using node 127.0.0.1:7000)
M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000
slots:0-5959,10922-11422 (6461 slots) master
M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002
slots:11423-16383 (4961 slots) master
S: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005
slots: (0 slots) slave
M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001
slots:5960-10921 (4962 slots) master
S: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003
slots: (0 slots) slave
S: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004
slots: (0 slots) slave
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

根据检查结果显示， 集群运作正常。

需要注意的就是， 在三个主节点中， 节点 `127.0.0.1:7000` 包含了 `6461` 个哈希槽， 而节点 `127.0.0.1:7001` 和节点 `127.0.0.1:7002` 都只包含了 `4961` 个哈希槽， 因为后两者都将自己的 `500` 个哈希槽移动到了节点 `127.0.0.1:7000` 。

# 一个更有趣的示例应用

我们在前面使用的示例程序 `example.rb` 并不是十分有趣， 因为它只是不断地对集群进行写入， 但并不检查写入结果是否正确。 比如说， 集群可能会错误地将 `example.rb` 发送的所有 SET 命令都改成了 `SET foo 42` ， 但因为 `example.rb` 并不检查写入后的值， 所以它不会意识到集群实际上写入的值是错误的。

因为这个原因， redis-rb-cluster 项目包含了一个名为 consistency-test.rb 的示例应用， 这个应用比起 `example.rb` 有趣得多： 它创建了多个计数器（默认为 `1000` 个）， 并通过发送 INCR 命令来增加这些计数器的值。

在增加计数器值的同时， `consistency-test.rb` 还执行以下操作：

- 每次使用 INCR 命令更新一个计数器时， 应用会记录下计数器执行 INCR 命令之后应该有的值。 举个例子， 如果计数器的起始值为 `0` ， 而这次是程序第 `50` 次向它发送 INCR 命令， 那么计数器的值应该是 `50` 。
- 在每次发送 INCR 命令之前， 程序会随机从集群中读取一个计数器的值， 并将它与自己记录的值进行对比， 看两个值是否相同。

换句话说， 这个程序是一个一致性检查器（consistency checker）： 如果集群在执行 INCR 命令的过程中， 丢失了某条 INCR 命令， 又或者多执行了某条客户端没有确认到的 INCR 命令， 那么检查器将察觉到这一点 —— 在前一种情况中， `consistency-test.rb` 记录的计数器值将比集群记录的计数器值要大； 而在后一种情况中， `consistency-test.rb` 记录的计数器值将比集群记录的计数器值要小。

运行 `consistency-test` 程序将产生类似以下的输出：

```
$ ruby consistency-test.rb
925 R (0 err) | 925 W (0 err) |
5030 R (0 err) | 5030 W (0 err) |
9261 R (0 err) | 9261 W (0 err) |
13517 R (0 err) | 13517 W (0 err) |
17780 R (0 err) | 17780 W (0 err) |
22025 R (0 err) | 22025 W (0 err) |
25818 R (0 err) | 25818 W (0 err) |
```

每行输出都打印了程序执行的读取次数和写入次数， 以及执行操作的过程中因为集群不可用而产生的错误数。

如果程序察觉了不一致的情况出现， 它将在输出行的末尾显式不一致的详细情况。

比如说， 如果我们在 `consistency-test.rb` 运行的过程中， 手动修改某个计数器的值：

```
$ redis 127.0.0.1:7000> set key_217 0
OK
```

那么 `consistency-test.rb` 将向我们报告不一致情况：

```
(in the other tab I see...)

94774 R (0 err) | 94774 W (0 err) |
98821 R (0 err) | 98821 W (0 err) |
102886 R (0 err) | 102886 W (0 err) | 114 lost |
107046 R (0 err) | 107046 W (0 err) | 114 lost |
```

在我们修改计数器值的时候， 计数器的正确值是 `114` （执行了 `114` 次 INCR 命令）， 因为我们将计数器的值设成了 `0` ， 所以 `consistency-test.rb` 会向我们报告说丢失了 `114` 个 INCR 命令。

因为这个示例程序具有一致性检查功能， 所以我们用它来测试 Redis 集群的故障转移操作。

# 故障转移测试

Note

在执行本节操作的过程中， 请一直运行 `consistency-test` 程序。

要触发一次故障转移， 最简单的办法就是令集群中的某个主节点进入下线状态。

首先用以下命令列出集群中的所有主节点：

```
$ redis-cli -p 7000 cluster nodes | grep master
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385482984082 0 connec
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 master - 0 1385482983582 0 connec
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-
```

通过命令输出，我们知道端口号为 `7000` 、 `7001` 和 `7002` 的节点都是主节点，然后我们可以通过向端口号为 `7002` 的主节点发送 DEBUG SEGFAULT 命令，让这个主节点崩溃：

```
$ redis-cli -p 7002 debug segfault
Error: Server closed the connection
```

现在，切换到运行着 `consistency-test` 的标签页，可以看到，`consistency-test` 在 `7002` 下线之后的一段时间里将产生大量的错误警告信息：

```
18849 R (0 err) | 18849 W (0 err) |
23151 R (0 err) | 23151 W (0 err) |
27302 R (0 err) | 27302 W (0 err) |

... many error warnings here ...

29659 R (578 err) | 29660 W (577 err) |
33749 R (578 err) | 33750 W (577 err) |
37918 R (578 err) | 37919 W (577 err) |
42077 R (578 err) | 42078 W (577 err) |
```

从 `consistency-test` 的这段输出可以看到，集群在执行故障转移期间，总共丢失了 `578` 个读命令和 `577` 个写命令，但是并没有产生任何数据不一致。

这听上去可能有点奇怪，因为在教程的开头我们提到过，Redis 使用的是异步复制，在执行故障转移期间，集群可能会丢失写命令。

但是在实际上，丢失命令的情况并不常见，因为 Redis 几乎是同时执行将命令回复发送给客户端，以及将命令复制给从节点这两个操作，所以实际上造成命令丢失的时间窗口是非常小的。

不过，尽管出现的几率不高，但丢失命令的情况还是有可能会出现的，所以我们对 Redis 集群不能提供强一致性的这一描述仍然是正确的。

现在，让我们使用 `cluster nodes` 命令，查看集群在执行故障转移操作之后，主从节点的布局情况：

```
$ redis-cli -p 7000 cluster nodes
3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b
a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d68
97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-
3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 1385503419023 3 connec
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385503417005 0 connec
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b
```

我重启了之前下线的 `127.0.0.1:7002` 节点， 该节点已经从原来的主节点变成了从节点， 而现在集群中的三个主节点分别是 `127.0.0.1:7000` 、 `127.0.0.1:7001` 和 `127.0.0.1:7005` ，其中 `127.0.0.1:7005` 就是因为 `127.0.0.1:7002` 下线而变成主节点的。

`cluster nodes` 命令的输出有点儿复杂， 它的每一行都是由以下信息组成的：

- 节点 ID ：例如 `3fc783611028b1707fd65345e763befb36454d73` 。
- `ip:port` ：节点的 IP 地址和端口号， 例如 `127.0.0.1:7000` ， 其中 `:0` 表示的是客户端当前连接的 IP 地址和端口号。
- `flags` ：节点的角色（例如 `master` 、 `slave` 、 `myself` ）以及状态（例如 `fail` ，等等）。
- 如果节点是一个从节点的话， 那么跟在 `flags` 之后的将是主节点的节点 ID ： 例如 `127.0.0.1:7002` 的主节点的节点 ID 就是 `3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e` 。
- 集群最近一次向节点发送 PING 命令之后， 过去了多长时间还没接到回复。
- 节点最近一次返回 `PONG` 回复的时间。
- 节点的配置纪元（configuration epoch）：详细信息请参考 Redis 集群规范 。
- 本节点的网络连接情况：例如 `connected` 。
- 节点目前包含的槽：例如 `127.0.0.1:7001` 目前包含号码为 `5960` 至 `10921` 的哈希槽。

## 添加新节点到集群

根据新添加节点的种类， 我们需要用两种方法来将新节点添加到集群里面：

- 如果要添加的新节点是一个主节点， 那么我们需要创建一个空节点（empty node）， 然后将某些哈希桶移动到这个空节点里面。
- 另一方面， 如果要添加的新节点是一个从节点， 那么我们需要将这个新节点设置为集群中某个节点的复制品（replica）。

本节将对以上两种情况进行介绍， 首先介绍主节点的添加方法， 然后再介绍从节点的添加方法。

无论添加的是那种节点， 第一步要做的总是添加一个空节点。

我们可以继续使用之前启动 `127.0.0.1:7000` 、 `127.0.0.1:7001` 等节点的方法， 创建一个端口号为 `7006` 的新节点， 使用的配置文件也和之前一样， 只是记得要将配置中的端口号改为 `7000` 。

以下是启动端口号为 `7006` 的新节点的详细步骤：

1. 在终端里创建一个新的标签页。
2. 进入 `cluster-test` 文件夹。
3. 创建并进入 `7006` 文件夹。
4. 将 `redis.conf` 文件复制到 `7006` 文件夹里面，然后将配置中的端口号选项改为 `7006` 。
5. 使用命令 `../../redis-server redis.conf` 启动节点。

如果一切正常， 那么节点应该会正确地启动。

接下来， 执行以下命令， 将这个新节点添加到集群里面：

```
./redis-trib.rb add-node 127.0.0.1:7006 127.0.0.1:7000
```

命令中的 `add-node` 表示我们要让 `redis-trib` 将一个节点添加到集群里面， `add-node` 之后跟着的是新节点的 IP 地址和端口号， 再之后跟着的是集群中任意一个已存在节点的 IP 地址和端口号， 这里我们使用的是 `127.0.0.1:7000` 。

通过 `cluster nodes` 命令， 我们可以确认新节点 `127.0.0.1:7006` 已经被添加到集群里面了：

```
redis 127.0.0.1:7006> cluster nodes
3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385543178575 0 connec
3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b
f093c80dde814da99c5cf72a7dd01590792b783b :0 myself,master - 0 0 0 connected
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b
a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d68
97a3a64667477371c4479320d683e4c8db5858b1 127.0.0.1:7000 master - 0 1385543179080 0 connec
3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 1385543177568 3 connec
```

新节点现在已经连接上了集群， 成为集群的一份子， 并且可以对客户端的命令请求进行转向了， 但是和其他主节点相比， 新节点还有两点区别：

- 新节点没有包含任何数据， 因为它没有包含任何哈希桶。
- 尽管新节点没有包含任何哈希桶， 但它仍然是一个主节点， 所以在集群需要将某个从节点升级为新的主节点时， 这个新节点不会被选中。

接下来， 只要使用 `redis-trib` 程序， 将集群中的某些哈希桶移动到新节点里面， 新节点就会成为真正的主节点了。

因为使用 `redis-trib` 移动哈希桶的方法在前面已经介绍过， 所以这里就不再重复介绍了。

现在， 让我们来看看， 将一个新节点转变为某个主节点的复制品（也即是从节点）的方法。

举个例子， 如果我们打算让新节点成为 `127.0.0.1:7005` 的从节点， 那么我们只要用客户端连接上新节点， 然后执行以下命令就可以了：

```
redis 127.0.0.1:7006> cluster replicate 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e
```

其中命令提供的 `3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e` 就是主节点 `127.0.0.1:7005` 的节点 ID 。

执行 `cluster replicate` 命令之后， 我们可以使用以下命令来确认 `127.0.0.1:7006` 已经成为了 ID 为 `3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e` 的节点的从节点：

```
$ redis-cli -p 7000 cluster nodes | grep slave | grep 3c3a0c74aae0b56170ccb03a76b60cfe7dc
f093c80dde814da99c5cf72a7dd01590792b783b 127.0.0.1:7006 slave 3c3a0c74aae0b56170ccb03a76b
2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b
```

`3c3a0c...` 现在有两个从节点， 一个从节点的端口号为 `7002` ， 而另一个从节点的端口号为 `7006` 。

## 移除一个节点

未完待续。

# Redis 集群规范

Note

本文档翻译自 http://redis.io/topics/cluster-spec 。

## 引言

这个文档是正在开发中的 Redis 集群功能的规范（specification）文档， 文档分为两个部分：

- 第一部分介绍目前已经在 `unstable` 分支中实现了的那些功能。
- 第二部分介绍目前仍未实现的那些功能。

文档各个部分的内容可能会随着集群功能的设计修改而发生改变， 其中， 未实现功能发生修改的几率比已实现功能发生修改的几率要高。

这个规范包含了编写客户端库（client library）所需的全部知识， 不过请注意， 这里列出的一部分细节可能会在未来发生变化。

## 什么是 Redis 集群？

Redis 集群是一个分布式（distributed）、容错（fault-tolerant）的 Redis 实现， 集群可以使用的功能是普通单机 Redis 所能使用的功能的一个子集（subset）。

Redis 集群中不存在中心（central）节点或者代理（proxy）节点， 集群的其中一个主要设计目标是达到线性可扩展性（linear scalability）。

Redis 集群为了保证一致性（consistency）而牺牲了一部分容错性： 系统会在保证对网络断线（net split）和节点失效（node failure）具有有限（limited）抵抗力的前提下， 尽可能地保持数据的一致性。

Note

集群将节点失效视为网络断线的其中一种特殊情况。

集群的容错功能是通过使用主节点（master）和从节点（slave）两种角色（role）的节点（node）来实现的：

- 主节点和从节点使用完全相同的服务器实现， 它们的功能（functionally）也完全一样， 但从节点通常仅用于替换失效的主节点。
- 不过， 如果不需要保证“先写入，后读取”操作的一致性（read-after-write consistency）， 那么可以使用从节点来执行只读查询。

# Redis 集群实现的功能子集

Redis 集群实现了单机 Redis 中， 所有处理单个数据库键的命令。

针对多个数据库键的复杂计算操作， 比如集合的并集操作、合集操作没有被实现， 那些理论上需要使用多个节点的多个数据库键才能完成的命令也没有被实现。

在将来， 用户也许可以通过 MIGRATE COPY 命令， 在集群的计算节点（computation node）中执行针对多个数据库键的只读操作， 但集群本身不会去实现那些需要将多个数据库键在多个节点中移来移去的复杂多键命令。

Redis 集群不像单机 Redis 那样支持多数据库功能， 集群只使用默认的 `0` 号数据库， 并且不能使用 SELECT 命令。

# Redis 集群协议中的客户端和服务器

Redis 集群中的节点有以下责任：

- 持有键值对数据。
- 记录集群的状态，包括键到正确节点的映射（mapping keys to right nodes）。
- 自动发现其他节点，识别工作不正常的节点，并在有需要时，在从节点中选举出新的主节点。

为了执行以上列出的任务， 集群中的每个节点都与其他节点建立起了“集群连接（cluster bus）”， 该连接是一个 TCP 连接， 使用二进制协议进行通讯。

节点之间使用 Gossip 协议 来进行以下工作：

- 传播（propagate）关于集群的信息，以此来发现新的节点。
- 向其他节点发送 `PING` 数据包，以此来检查目标节点是否正常运作。
- 在特定事件发生时，发送集群信息。

除此之外， 集群连接还用于在集群中发布或订阅信息。

因为集群节点不能代理（proxy）命令请求， 所以客户端应该在节点返回 `-MOVED` 或者 `-ASK` 转向（redirection）错误时， 自行将命令请求转发至其他节点。

因为客户端可以自由地向集群中的任何一个节点发送命令请求， 并可以在有需要时， 根据转向错误所提供的信息， 将命令转发至正确的节点， 所以在理论上来说， 客户端是无须保存集群状态信息的。

不过， 如果客户端可以将键和节点之间的映射信息保存起来， 可以有效地减少可能出现的转向次数， 籍此提升命令执行的效率。

# 键分布模型

Redis 集群的键空间被分割为 `16384` 个槽（slot）， 集群的最大节点数量也是 `16384` 个。

Note

推荐的最大节点数量为 1000 个左右。

每个主节点都负责处理 `16384` 个哈希槽的其中一部分。

当我们说一个集群处于“稳定”（stable）状态时， 指的是集群没有在执行重配置（reconfiguration）操作， 每个哈希槽都只由一个节点进行处理。

Note

重配置指的是将某个/某些槽从一个节点移动到另一个节点。

Note

一个主节点可以有任意多个从节点， 这些从节点用于在主节点发生网络断线或者节点失效时， 对主节点进行替换。

以下是负责将键映射到槽的算法：

```
HASH_SLOT = CRC16(key) mod 16384
```

以下是该算法所使用的参数：

- 算法的名称: XMODEM (又称 ZMODEM 或者 CRC-16/ACORN)
- 结果的长度: 16 位
- 多项数（poly）: 1021 (也即是 `x16 + x12 + x5 + 1`)
- 初始化值: `0000`
- 反射输入字节（Reflect Input byte）： `False`
- 发射输出 CRC （Reflect Output CRC）： `False`
- 用于 CRC 输出值的异或常量（Xor constant to output CRC）： `0000`
- 该算法对于输入 `"123456789"` 的输出: `31C3`

附录 A 中给出了集群所使用的 CRC16 算法的实现。

CRC16 算法所产生的 16 位输出中的 14 位会被用到。

在我们的测试中， CRC16 算法可以很好地将各种不同类型的键平稳地分布到 `16384` 个槽里面。

# 集群节点属性

每个节点在集群中都有一个独一无二的 ID， 该 ID 是一个十六进制表示的 160 位随机数， 在节点第一次启动时由 `/dev/urandom` 生成。

节点会将它的 ID 保存到配置文件， 只要这个配置文件不被删除， 节点就会一直沿用这个 ID 。

节点 ID 用于标识集群中的每个节点。 一个节点可以改变它的 IP 和端口号， 而不改变节点 ID 。 集群可以自动识别出 IP/端口号的变化， 并将这一信息通过 Gossip 协议广播给其他节点知道。

以下是每个节点都有的关联信息， 并且节点会将这些信息发送给其他节点：

- 节点所使用的 IP 地址和 TCP 端口号。
- 节点的标志（flags）。
- 节点负责处理的哈希槽。
- 节点最近一次使用集群连接发送 `PING` 数据包（packet）的时间。
- 节点最近一次在回复中接收到 `PONG` 数据包的时间。
- 集群将该节点标记为下线的时间。
- 该节点的从节点数量。
- 如果该节点是从节点的话，那么它会记录主节点的节点 ID 。 如果这是一个主节点的话，那么主节点 ID 这一栏的值为 `0000000` 。

以上信息的其中一部分可以通过向集群中的任意节点（主节点或者从节点都可以）发送 `CLUSTER NODES` 命令来获得。

以下是一个向集群中的主节点发送 `CLUSTER NODES` 命令的例子， 该集群由三个节点组成：

```
$ redis-cli cluster nodes
d1861060fe6a534d42d8a19aeb36600e18785e04 :0 myself - 0 1318428930 connected 0-1364
3886e65cc906bfd9b1f7e7bde468726a052d1dae 127.0.0.1:6380 master - 1318428930 1318428931 co
d289c575dcbc4bdd2931585fd4339089e461a27d 127.0.0.1:6381 master - 1318428931 1318428931 co
```

在上面列出的三行信息中， 从左到右的各个域分别是： 节点 ID ， IP 地址和端口号， 标志（flag）， 最后发送 `PING` 的时间， 最后接收 `PONG` 的时间， 连接状态， 节点负责处理的槽。

# 节点握手（已实现）

节点总是应答（accept）来自集群连接端口的连接请求， 并对接收到的 `PING` 数据包进行回复， 即使这个 `PING` 数据包来自不可信的节点。

然而， 除了 `PING` 之外， 节点会拒绝其他所有并非来自集群节点的数据包。

要让一个节点承认另一个节点同属于一个集群， 只有以下两种方法：

- 一个节点可以通过向另一个节点发送 `MEET` 信息，来强制让接收信息的节点承认发送信息的节点为集群中的一份子。一个节点仅在管理员显式地向它发送 `CLUSTER MEET ip port` 命令时，才会向另一个节点发送 `MEET` 信息。
- 另外，如果一个可信节点向另一个节点传播第三者节点的信息，那么接收信息的那个节点也会将第三者节点识别为集群中的一份子。也即是说，如果 A 认识 B，B 认识 C，并且 B 向 A 传播关于 C 的信息，那么 A 也会将 C 识别为集群中的一份子，并尝试连接 C。

这意味着如果我们将一个/一些新节点添加到一个集群中，那么这个/这些新节点最终会和集群中已有的其他所有节点连接起来。

这说明只要管理员使用 `CLUSTER MEET` 命令显式地指定了可信关系，集群就可以自动发现其他节点。

这种节点识别机制通过防止不同的 Redis 集群因为 IP 地址变更或者其他网络事件的发生而产生意料之外的联合（mix），从而使得集群更具健壮性。

当节点的网络连接断开时，它会主动连接其他已知的节点。

# MOVED 转向

一个 Redis 客户端可以向集群中的任意节点（包括从节点）发送命令请求。节点会对命令请求进行分析，如果该命令是集群可以执行的命令，那么节点会查找这个命令所要处理的键所在的槽。

如果要查找的哈希槽正好就由接收到命令的节点负责处理，那么节点就直接执行这个命令。

另一方面，如果所查找的槽不是由该节点处理的话，节点将查看自身内部所保存的哈希槽到节点 ID 的映射记录，并向客户端回复一个 `MOVED` 错误。

以下是一个 `MOVED` 错误的例子：

```
GET x

-MOVED 3999 127.0.0.1:6381
```

错误信息包含键 `x` 所属的哈希槽 `3999`，以及负责处理这个槽的节点的 IP 和端口号 `127.0.0.1:6381`。客户端需要根据这个 IP 和端口号，向所属的节点重新发送一次 GET 命令请求。

注意，即使客户端在重新发送 GET 命令之前，等待了非常久的时间，以至于集群又再次更改了配置，使得节点 `127.0.0.1:6381` 已经不再处理槽 `3999`，那么当客户端向节点 `127.0.0.1:6381` 发送 GET 命令的时候，节点将再次向客户端返回 `MOVED` 错误，指示现在负责处理槽 `3999` 的节点。

虽然我们用 ID 来标识集群中的节点， 但是为了让客户端的转向操作尽可能地简单， 节点在 `MOVED` 错误中直接返回目标节点的 IP 和端口号， 而不是目标节点的 ID 。

虽然不是必须的， 但一个客户端应该记录（memorize）下“槽 `3999` 由节点 `127.0.0.1:6381` 负责处理“这一信息， 这样当再次有命令需要对槽 `3999` 执行时， 客户端就可以加快寻找正确节点的速度。

注意， 当集群处于稳定状态时， 所有客户端最终都会保存有一个哈希槽至节点的映射记录（map of hash slots to nodes）， 使得集群非常高效： 客户端可以直接向正确的节点发送命令请求， 无须转向、代理或者其他任何可能发生单点故障（single point failure）的实体（entiy）。

除了 `MOVED` 转向错误之外， 一个客户端还应该可以处理稍后介绍的 `ASK` 转向错误。

# 集群在线重配置（live reconfiguration）

Redis 集群支持在集群运行的过程中添加或者移除节点。

实际上， 节点的添加操作和节点的删除操作可以抽象成同一个操作， 那就是， 将哈希槽从一个节点移动到另一个节点：

- 添加一个新节点到集群， 等于将其他已存在节点的槽移动到一个空白的新节点里面。
- 从集群中移除一个节点， 等于将被移除节点的所有槽移动到集群的其他节点上面去。

因此， 实现 Redis 集群在线重配置的核心就是将槽从一个节点移动到另一个节点的能力。 因为一个哈希槽实际上就是一些键的集合， 所以 Redis 集群在重哈希（rehash）时真正要做的， 就是将一些键从一个节点移动到另一个节点。

要理解 Redis 集群如何将槽从一个节点移动到另一个节点， 我们需要对 `CLUSTER` 命令的各个子命令进行介绍， 这些命理负责管理集群节点的槽转换表（slots translation table）。

以下是 `CLUSTER` 命令可用的子命令：

- `CLUSTER ADDSLOTS slot1 [slot2] ... [slotN]`
- `CLUSTER DELSLOTS slot1 [slot2] ... [slotN]`
- `CLUSTER SETSLOT slot NODE node`
- `CLUSTER SETSLOT slot MIGRATING node`
- `CLUSTER SETSLOT slot IMPORTING node`

最开头的两条命令 `ADDSLOTS` 和 `DELSLOTS` 分别用于向节点指派（assign）或者移除节点， 当槽被指派或者移除之后， 节点会将这一信息通过 Gossip 协议传播到整个集群。 `ADDSLOTS` 命令通常在新创建集群时， 作为一种快速地将各个槽指派给各个节点的手段来使用。

`CLUSTER SETSLOT slot NODE node` 子命令可以将指定的槽 `slot` 指派给节点 `node` 。

至于 `CLUSTER SETSLOT slot MIGRATING node` 命令和 `CLUSTER SETSLOT slot IMPORTING node` 命令，前者用于将给定节点 `node` 中的槽 `slot` 迁移出节点，而后者用于将给定槽 `slot` 导入到节点 `node` ：

- 当一个槽被设置为 `MIGRATING` 状态时，原来持有这个槽的节点仍然会继续接受关于这个槽的命令请求，但只有命令所处理的键仍然存在于节点时，节点才会处理这个命令请求。

  如果命令所使用的键不存在与该节点，那么节点将向客户端返回一个 `-ASK` 转向（redirection）错误，告知客户端，要将命令请求发送到槽的迁移目标节点。

- 当一个槽被设置为 `IMPORTING` 状态时，节点仅在接收到 `ASKING` 命令之后，才会接受关于这个槽的命令请求。

  如果客户端没有向节点发送 `ASKING` 命令，那么节点会使用 `-MOVED` 转向错误将命令请求转向至真正负责处理这个槽的节点。

上面关于 `MIGRATING` 和 `IMPORTING` 的说明有些难懂，让我们用一个实际的实例来说明一下。

假设现在，我们有 A 和 B 两个节点，并且我们想将槽 `8` 从节点 A 移动到节点 B ，于是我们：

- 向节点 B 发送命令 `CLUSTER SETSLOT 8 IMPORTING A`
- 向节点 A 发送命令 `CLUSTER SETSLOT 8 MIGRATING B`

每当客户端向其他节点发送关于哈希槽 `8` 的命令请求时，这些节点都会向客户端返回指向节点 A 的转向信息：

- 如果命令要处理的键已经存在于槽 `8` 里面，那么这个命令将由节点 A 处理。
- 如果命令要处理的键未存在于槽 `8` 里面（比如说，要向槽添加一个新的键），那么这个命令由节点 B 处理。

这种机制将使得节点 A 不再创建关于槽 `8` 的任何新键。

与此同时，一个特殊的客户端 `redis-trib` 以及 Redis 集群配置程序（configuration utility）会将节点 A 中槽 `8` 里面的键移动到节点 B 。

键的移动操作由以下两个命令执行：

```
CLUSTER GETKEYSINSLOT slot count
```

上面的命令会让节点返回 `count` 个 `slot` 槽中的键，对于命令所返回的每个键，`redis-trib` 都会向节点 A 发送一条 MIGRATE 命令，该命令会将所指定的键原子地（atomic）从节点 A 移动到节点 B （在移动键期间，两个节点都会处于阻塞状态，以免出现竞争条件）。

以下为 MIGRATE 命令的运作原理：

```
MIGRATE target_host target_port key target_database id timeout
```

执行 MIGRATE 命令的节点会连接到 `target` 节点， 并将序列化后的 `key` 数据发送给 `target` ， 一旦 `target` 返回 `OK` ， 节点就将自己的 `key` 从数据库中删除。

从一个外部客户端的视角来看， 在某个时间点上， 键 `key` 要么存在于节点 A ， 要么存在于节点 B ， 但不会同时存在于节点 A 和节点 B 。

因为 Redis 集群只使用 `0` 号数据库， 所以当 MIGRATE 命令被用于执行集群操作时， `target_database` 的值总是 `0` 。

`target_database` 参数的存在是为了让 MIGRATE 命令成为一个通用命令， 从而可以作用于集群以外的其他功能。

我们对 MIGRATE 命令做了优化， 使得它即使在传输包含多个元素的列表键这样的复杂数据时， 也可以保持高效。

不过， 尽管 MIGRATE 非常高效， 对一个键非常多、并且键的数据量非常大的集群来说， 集群重配置还是会占用大量的时间， 可能会导致集群没办法适应那些对于响应时间有严格要求的应用程序。

# ASK 转向

在之前介绍 `MOVED` 转向的时候， 我们说除了 `MOVED` 转向之外， 还有另一种 `ASK` 转向。

当节点需要让一个客户端长期地（permanently）将针对某个槽的命令请求发送至另一个节点时， 节点向客户端返回 `MOVED` 转向。

另一方面， 当节点需要让客户端仅仅在下一个命令请求中转向至另一个节点时， 节点向客户端返回 `ASK` 转向。

比如说， 在我们上一节列举的槽 `8` 的例子中， 因为槽 `8` 所包含的各个键分散在节点 A 和节点 B 中， 所以当客户端在节点 A 中没找到某个键时， 它应该转向到节点 B 中去寻找， 但是这种转向应该仅仅影响一次命令查询， 而不是让客户端每次都直接去查找节点 B ： 在节点 A 所持有的属于槽 `8` 的键没有全部被迁移到节点 B 之前， 客户端应该先访问节点 A ， 然后再访问节点 B 。

因为这种转向只针对 `16384` 个槽中的其中一个槽， 所以转向对集群造成的性能损耗属于可接受的范围。

因为上述原因， 如果我们要在查找节点 A 之后， 继续查找节点 B ， 那么客户端在向节点 B 发送命令请求之前， 应该先发送一个 `ASKING` 命令， 否则这个针对带有 `IMPORTING` 状态的槽的命令请求将被节点 B 拒绝执行。

接收到客户端 `ASKING` 命令的节点将为客户端设置一个一次性的标志（flag）， 使得客户端可以执行一次针对 `IMPORTING` 状态的槽的命令请求。

从客户端的角度来看， `ASK` 转向的完整语义（semantics）如下：

- 如果客户端接收到 `ASK` 转向， 那么将命令请求的发送对象调整为转向所指定的节点。
- 先发送一个 `ASKING` 命令，然后再发送真正的命令请求。
- 不必更新客户端所记录的槽 `8` 至节点的映射： 槽 `8` 应该仍然映射到节点 A， 而不是节点 B 。

一旦节点 A 针对槽 `8` 的迁移工作完成， 节点 A 在再次收到针对槽 `8` 的命令请求时， 就会向客户端返回 `MOVED` 转向， 将关于槽 `8` 的命令请求长期地转向到节点 B 。

注意， 即使客户端出现 Bug ， 过早地将槽 `8` 映射到了节点 B 上面， 但只要这个客户端不发送 `ASKING` 命令， 客户端发送命令请求的时候就会遇上 `MOVED` 错误， 并将它转向回节点 A 。

# 容错

## 节点失效检测

以下是节点失效检查的实现方法：

- 当一个节点向另一个节点发送 PING 命令， 但是目标节点未能在给定的时限内返回 PING 命令的回复时， 那么发送命令的节点会将目标节点标记为 `PFAIL` （possible failure，可能已失效）。

  等待 PING 命令回复的时限称为“节点超时时限（node timeout）”， 是一个节点选项（node-wise setting）。

- 每次当节点对其他节点发送 PING 命令的时候， 它都会随机地广播三个它所知道的节点的信息， 这些信息里面的其中一项就是说明节点是否已经被标记为 `PFAIL` 或者 `FAIL` 。

- 当节点接收到其他节点发来的信息时， 它会记下那些被其他节点标记为失效的节点。 这称为失效报告（failure report）。

- 如果节点已经将某个节点标记为 `PFAIL` ， 并且根据节点所收到的失效报告显式， 集群中的大部分其他主节点也认为那个节点进入了失效状态， 那么节点会将那个失效节点的状态标记为 `FAIL` 。

- 一旦某个节点被标记为 `FAIL` ， 关于这个节点已失效的信息就会被广播到整个集群， 所有接收到这条信息的节点都会将失效节点标记为 `FAIL` 。

简单来说， 一个节点要将另一个节点标记为失效， 必须先询问其他节点的意见， 并且得到大部分主节点的同意才行。

因为过期的失效报告会被移除， 所以主节点要将某个节点标记为 `FAIL` 的话， 必须以最近接收到的失效报告作为根据。

在以下两种情况中， 节点的 `FAIL` 状态会被移除：

- 如果被标记为 `FAIL` 的是从节点， 那么当这个节点重新上线时， `FAIL` 标记就会被移除。

  保持（retaning）从节点的 `FAIL` 状态是没有意义的， 因为它不处理任何槽， 一个从节点是否处于 `FAIL` 状态， 决定了这个从节点在有需要时能否被提升为主节点。

- 如果一个主节点被打上 `FAIL` 标记之后， 经过了节点超时时限的四倍时间， 再加上十秒钟之后， 针对这个主节点的槽的故障转移操作仍未完成， 并且这个主节点已经重新上线的话， 那么移除对这个节点的 `FAIL` 标记。

在第二种情况中， 如果故障转移未能顺利完成， 并且主节点重新上线， 那么集群就继续使用原来的主节点， 从而免去管理员介入的必要。

## 集群状态检测（已部分实现）

每当集群发生配置变化时（可能是哈希槽更新，也可能是某个节点进入失效状态）， 集群中的每个节点都会对它所知道的节点进行扫描（scan）。

一旦配置处理完毕， 集群会进入以下两种状态的其中一种：

- `FAIL` ： 集群不能正常工作。 当集群中有某个节点进入失效状态时， 集群不能处理任何命令请求， 对于每个命令请求， 集群节点都返回错误回复。
- `OK` ： 集群可以正常工作， 负责处理全部 `16384` 个槽的节点中， 没有一个节点被标记为 `FAIL` 状态。

这说明即使集群中只有一部分哈希槽不能正常使用， 整个集群也会停止处理任何命令。

不过节点从出现问题到被标记为 `FAIL` 状态的这段时间里， 集群仍然会正常运作， 所以集群在某些时候， 仍然有可能只能处理针对 `16384` 个槽的其中一个子集的命令请求。

以下是集群进入 `FAIL` 状态的两种情况：

1. 至少有一个哈希槽不可用，因为负责处理这个槽的节点进入了 `FAIL` 状态。
2. 集群中的大部分主节点都进入下线状态。当大部分主节点都进入 `PFAIL` 状态时，集群也会进入 `FAIL` 状态。

第二个检查是必须的， 因为要将一个节点从 `PFAIL` 状态改变为 `FAIL` 状态， 必须要有大部分主节点进行投票表决， 但是， 当集群中的大部分主节点都进入失效状态时， 单凭一个两个节点是没有办法将一个节点标记为 `FAIL` 状态的。

因此， 有了第二个检查条件， 只要集群中的大部分主节点进入了下线状态， 那么集群就可以在不请求这些主节点的意见下， 将某个节点判断为 `FAIL` 状态， 从而让整个集群停止处理命令请求。

## 从节点选举

一旦某个主节点进入 `FAIL` 状态， 如果这个主节点有一个或多个从节点存在， 那么其中一个从节点会被升级为新的主节点， 而其他从节点则会开始对这个新的主节点进行复制。

新的主节点由已下线主节点属下的所有从节点中自行选举产生， 以下是选举的条件：

- 这个节点是已下线主节点的从节点。
- 已下线主节点负责处理的槽数量非空。
- 从节点的数据被认为是可靠的， 也即是， 主从节点之间的复制连接（replication link）的断线时长不能超过节点超时时限（node timeout）乘以 `REDIS_CLUSTER_SLAVE_VALIDITY_MULT` 常量得出的积。

如果一个从节点满足了以上的所有条件， 那么这个从节点将向集群中的其他主节点发送授权请求， 询问它们， 是否允许自己（从节点）升级为新的主节点。

如果发送授权请求的从节点满足以下属性， 那么主节点将向从节点返回 `FAILOVER_AUTH_GRANTED` 授权， 同意从节点的升级要求：

- 发送授权请求的是一个从节点， 并且它所属的主节点处于 `FAIL` 状态。
- 在已下线主节点的所有从节点中， 这个从节点的节点 ID 在排序中是最小的。
- 这个从节点处于正常的运行状态： 它没有被标记为 `FAIL` 状态， 也没有被标记为 `PFAIL` 状态。

一旦某个从节点在给定的时限内得到大部分主节点的授权， 它就会开始执行以下故障转移操作：

- 通过 `PONG` 数据包（packet）告知其他节点， 这个节点现在是主节点了。
- 通过 `PONG` 数据包告知其他节点， 这个节点是一个已升级的从节点（promoted slave）。
- 接管（claiming）所有由已下线主节点负责处理的哈希槽。
- 显式地向所有节点广播一个 `PONG` 数据包， 加速其他节点识别这个节点的进度， 而不是等待定时的 `PING` / `PONG` 数据包。

所有其他节点都会根据新的主节点对配置进行相应的更新，特别地：

- 所有被新的主节点接管的槽会被更新。
- 已下线主节点的所有从节点会察觉到 `PROMOTED` 标志， 并开始对新的主节点进行复制。
- 如果已下线的主节点重新回到上线状态， 那么它会察觉到 `PROMOTED` 标志， 并将自身调整为现任主节点的从节点。

在集群的生命周期中， 如果一个带有 `PROMOTED` 标识的主节点因为某些原因转变成了从节点， 那么该节点将丢失它所带有的 `PROMOTED` 标识。

## 发布/订阅（已实现，但仍然需要改善）

在一个 Redis 集群中， 客户端可以订阅任意一个节点， 也可以向任意一个节点发送信息， 节点会对客户端所发送的信息进行转发。

在目前的实现中， 节点会将接收到的信息广播至集群中的其他所有节点， 在将来的实现中， 可能会使用 bloom filter 或者其他算法来优化这一操作。

## 附录 A： CRC16 算法的 ANSI 实现参考

```
/*
 * Copyright 2001-2010 Georges Menie (www.menie.org)
 * Copyright 2010 Salvatore Sanfilippo (adapted to Redis coding style)
 * All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 *     * Redistributions of source code must retain the above copyright
 *       notice, this list of conditions and the following disclaimer.
 *     * Redistributions in binary form must reproduce the above copyright
 *       notice, this list of conditions and the following disclaimer in the
 *       documentation and/or other materials provided with the distribution.
 *     * Neither the name of the University of California, Berkeley nor the
 *       names of its contributors may be used to endorse or promote products
 *       derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS BE LIABLE FOR ANY
 * DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
 * (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */

/* CRC16 implementation acording to CCITT standards.
 *
 * Note by @antirez: this is actually the XMODEM CRC 16 algorithm, using the
 * following parameters:
 *
 * Name                       : "XMODEM", also known as "ZMODEM", "CRC-16/ACORN"
 * Width                      : 16 bit
 * Poly                       : 1021 (That is actually x^16 + x^12 + x^5 + 1)
 * Initialization             : 0000
 * Reflect Input byte         : False
 * Reflect Output CRC         : False
 * Xor constant to output CRC : 0000
 * Output for "123456789"     : 31C3
 */

static const uint16_t crc16tab[256]= {
    0x0000,0x1021,0x2042,0x3063,0x4084,0x50a5,0x60c6,0x70e7,
    0x8108,0x9129,0xa14a,0xb16b,0xc18c,0xd1ad,0xe1ce,0xf1ef,
    0x1231,0x0210,0x3273,0x2252,0x52b5,0x4294,0x72f7,0x62d6,
```

```
    0x9339,0x8318,0xb37b,0xa35a,0xd3bd,0xc39c,0xf3ff,0xe3de,
    0x2462,0x3443,0x0420,0x1401,0x64e6,0x74c7,0x44a4,0x5485,
    0xa56a,0xb54b,0x8528,0x9509,0xe5ee,0xf5cf,0xc5ac,0xd58d,
    0x3653,0x2672,0x1611,0x0630,0x76d7,0x66f6,0x5695,0x46b4,
    0xb75b,0xa77a,0x9719,0x8738,0xf7df,0xe7fe,0xd79d,0xc7bc,
    0x48c4,0x58e5,0x6886,0x78a7,0x0840,0x1861,0x2802,0x3823,
    0xc9cc,0xd9ed,0xe98e,0xf9af,0x8948,0x9969,0xa90a,0xb92b,
    0x5af5,0x4ad4,0x7ab7,0x6a96,0x1a71,0x0a50,0x3a33,0x2a12,
    0xdbfd,0xcbdc,0xfbbf,0xeb9e,0x9b79,0x8b58,0xbb3b,0xab1a,
    0x6ca6,0x7c87,0x4ce4,0x5cc5,0x2c22,0x3c03,0x0c60,0x1c41,
    0xedae,0xfd8f,0xcdec,0xddcd,0xad2a,0xbd0b,0x8d68,0x9d49,
    0x7e97,0x6eb6,0x5ed5,0x4ef4,0x3e13,0x2e32,0x1e51,0x0e70,
    0xff9f,0xefbe,0xdfdd,0xcffc,0xbf1b,0xaf3a,0x9f59,0x8f78,
    0x9188,0x81a9,0xb1ca,0xa1eb,0xd10c,0xc12d,0xf14e,0xe16f,
    0x1080,0x00a1,0x30c2,0x20e3,0x5004,0x4025,0x7046,0x6067,
    0x83b9,0x9398,0xa3fb,0xb3da,0xc33d,0xd31c,0xe37f,0xf35e,
    0x02b1,0x1290,0x22f3,0x32d2,0x4235,0x5214,0x6277,0x7256,
    0xb5ea,0xa5cb,0x95a8,0x8589,0xf56e,0xe54f,0xd52c,0xc50d,
    0x34e2,0x24c3,0x14a0,0x0481,0x7466,0x6447,0x5424,0x4405,
    0xa7db,0xb7fa,0x8799,0x97b8,0xe75f,0xf77e,0xc71d,0xd73c,
    0x26d3,0x36f2,0x0691,0x16b0,0x6657,0x7676,0x4615,0x5634,
    0xd94c,0xc96d,0xf90e,0xe92f,0x99c8,0x89e9,0xb98a,0xa9ab,
    0x5844,0x4865,0x7806,0x6827,0x18c0,0x08e1,0x3882,0x28a3,
    0xcb7d,0xdb5c,0xeb3f,0xfb1e,0x8bf9,0x9bd8,0xabbb,0xbb9a,
    0x4a75,0x5a54,0x6a37,0x7a16,0x0af1,0x1ad0,0x2ab3,0x3a92,
    0xfd2e,0xed0f,0xdd6c,0xcd4d,0xbdaa,0xad8b,0x9de8,0x8dc9,
    0x7c26,0x6c07,0x5c64,0x4c45,0x3ca2,0x2c83,0x1ce0,0x0cc1,
    0xef1f,0xff3e,0xcf5d,0xdf7c,0xaf9b,0xbfba,0x8fd9,0x9ff8,
    0x6e17,0x7e36,0x4e55,0x5e74,0x2e93,0x3eb2,0x0ed1,0x1ef0
};

uint16_t crc16(const char *buf, int len) {
    int counter;
    uint16_t crc = 0;
    for (counter = 0; counter < len; counter++)
            crc = (crc<<8) ^ crc16tab[((crc>>8) ^ *buf++)&0x00FF];
    return crc;
}
```

# Redis 命令参考

# Key（键）

# DEL

### DEL key [key ...]

删除给定的一个或多个 `key` 。

不存在的 `key` 会被忽略。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为被删除的 `key` 的数量。删除单个字符串类型的 `key` ，时间复杂度为O(1)。删除单个列表、集合、有序集合或哈希表类型的 `key` ，时间复杂度为O(M)， `M` 为以上数据结构内的元素数量。

返回值：

被删除 `key` 的数量。

```
#  删除单个 key

redis> SET name huangz
OK

redis> DEL name
(integer) 1

# 删除一个不存在的 key

redis> EXISTS phone
(integer) 0

redis> DEL phone # 失败，没有 key 被删除
(integer) 0

# 同时删除多个 key

redis> SET name "redis"
OK

redis> SET type "key-value store"
OK

redis> SET website "redis.com"
OK

redis> DEL name type website
(integer) 3
```

# DUMP

**DUMP key**

序列化给定 `key` ，并返回被序列化的值，使用 *RESTORE* 命令可以将这个值反序列化为 Redis 键。

序列化生成的值有以下几个特点：

- 它带有 64 位的校验和，用于检测错误， *RESTORE* 在进行反序列化之前会先检查校验和。
- 值的编码格式和 RDB 文件保持一致。
- RDB 版本会被编码在序列化值当中，如果因为 Redis 的版本不同造成 RDB 格式不兼容，那么 Redis 会拒绝对这个值进行反序列化操作。

序列化的值不包括任何生存时间信息。

可用版本：

>= 2.6.0

时间复杂度：

查找给定键的复杂度为 O(1) ，对键进行序列化的复杂度为 O(N*M) ，其中 N 是构成 `key` 的 Redis 对象的数量，而 M 则是这些对象的平均大小。如果序列化的对象是比较小的字符串，那么复杂度为 O(1) 。

返回值：

如果 `key` 不存在，那么返回 `nil` 。否则，返回序列化之后的值。

```
redis> SET greeting "hello, dumping world!"
OK

redis> DUMP greeting
"\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1\xde"

redis> DUMP not-exists-key
(nil)
```

# EXISTS

### EXISTS key

检查给定 `key` 是否存在。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

若 `key` 存在，返回 `1` ，否则返回 `0` 。

```
redis> SET db "redis"
OK

redis> EXISTS db
(integer) 1

redis> DEL db
(integer) 1

redis> EXISTS db
(integer) 0
```

# EXPIRE

**EXPIRE key seconds**

为给定 `key` 设置生存时间，当 `key` 过期时(生存时间为 `0` )，它会被自动删除。

在 Redis 中，带有生存时间的 `key` 被称为『易失的』(volatile)。

生存时间可以通过使用 *DEL* 命令来删除整个 `key` 来移除，或者被 *SET* 和 *GETSET* 命令覆写(overwrite)，这意味着，如果一个命令只是修改(alter)一个带生存时间的 `key` 的值而不是用一个新的 `key` 值来代替(replace)它的话，那么生存时间不会被改变。

比如说，对一个 `key` 执行 *INCR* 命令，对一个列表进行 *LPUSH* 命令，或者对一个哈希表执行 *HSET* 命令，这类操作都不会修改 `key` 本身的生存时间。

另一方面，如果使用 *RENAME* 对一个 `key` 进行改名，那么改名后的 `key` 的生存时间和改名前一样。

*RENAME* 命令的另一种可能是，尝试将一个带生存时间的 `key` 改名成另一个带生存时间的 `another_key` ，这时旧的 `another_key` (以及它的生存时间)会被删除，然后旧的 `key` 会改名为 `another_key` ，因此，新的 `another_key` 的生存时间也和原本的 `key` 一样。

使用 *PERSIST* 命令可以在不删除 `key` 的情况下，移除 `key` 的生存时间，让 `key` 重新成为一个『持久的』(persistent) `key` 。

更新生存时间

可以对一个已经带有生存时间的 `key` 执行 *EXPIRE* 命令，新指定的生存时间会取代旧的生存时间。

过期时间的精确度

在 Redis 2.4 版本中，过期时间的延迟在 1 秒钟之内 —— 也即是，就算 `key` 已经过期，但它还是可能在过期之后一秒钟之内被访问到，而在新的 Redis 2.6 版本中，延迟被降低到 1 毫秒之内。

**Redis 2.1.3 之前的不同之处**

在 Redis 2.1.3 之前的版本中，修改一个带有生存时间的 `key` 会导致整个 `key` 被删除，这一行为是受当时复制(replication)层的限制而作出的，现在这一限制已经被修复。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功返回 `1` 。当 `key` 不存在或者不能为 `key` 设置生存时间时(比如在低于 2.1.3 版本的 Redis 中你尝试更新 `key` 的生存时间)，返回 `0` 。

```
redis> SET cache_page "www.google.com"
OK

redis> EXPIRE cache_page 30  # 设置过期时间为 30 秒
(integer) 1

redis> TTL cache_page    # 查看剩余生存时间
(integer) 23

redis> EXPIRE cache_page 30000   # 更新过期时间
(integer) 1

redis> TTL cache_page
(integer) 29996
```

# 模式：导航会话

假设你有一项 web 服务，打算根据用户最近访问的 N 个页面来进行物品推荐，并且假设用户停止阅览超过 60 秒，那么就清空阅览记录(为了减少物品推荐的计算量，并且保持推荐物品的新鲜度)。

这些最近访问的页面记录，我们称之为『导航会话』(Navigation session)，可以用 *INCR* 和 *RPUSH* 命令在 Redis 中实现它：每当用户阅览一个网页的时候，执行以下代码：

```
MULTI
    RPUSH pagewviews.user:<userid> http://.....
    EXPIRE pagewviews.user:<userid> 60
EXEC
```

如果用户停止阅览超过 60 秒，那么它的导航会话就会被清空，当用户重新开始阅览的时候，系统又会重新记录导航会话，继续进行物品推荐。

# EXPIREAT

**EXPIREAT key timestamp**

EXPIREAT 的作用和 *EXPIRE* 类似，都用于为 `key` 设置生存时间。

不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

如果生存时间设置成功，返回 `1` 。当 `key` 不存在或没办法设置生存时间，返回 `0` 。

```
redis> SET cache www.google.com
OK

redis> EXPIREAT cache 1355292000     # 这个 key 将在 2012.12.12 过期
(integer) 1

redis> TTL cache
(integer) 45081860
```

# KEYS

### KEYS pattern

查找所有符合给定模式 `pattern` 的 `key` 。

`KEYS *` 匹配数据库中所有 `key` 。 `KEYS h?llo` 匹配 `hello` ， `hallo` 和 `hxllo` 等。 `KEYS h*llo` 匹配 `hllo` 和 `heeeeello` 等。 `KEYS h[ae]llo` 匹配 `hello` 和 `hallo` ，但不匹配 `hillo` 。

特殊符号用 `\` 隔开

Warning

KEYS 的速度非常快，但在一个大的数据库中使用它仍然可能造成性能问题，如果你需要从一个数据集中查找特定的 `key` ，你最好还是用 Redis 的集合结构(set)来代替。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为数据库中 `key` 的数量。

返回值：

符合给定模式的 `key` 列表。

```
redis> MSET one 1 two 2 three 3 four 4  # 一次设置 4 个 key
OK

redis> KEYS *o*
1) "four"
2) "two"
3) "one"

redis> KEYS t??
1) "two"

redis> KEYS t[w]*
1) "two"

redis> KEYS *  # 匹配数据库内所有 key
1) "four"
2) "three"
3) "two"
4) "one"
```

# MIGRATE

**MIGRATE host port key destination-db timeout [COPY] [REPLACE]**

将 `key` 原子性地从当前实例传送到目标实例的指定数据库上，一旦传送成功， `key` 保证会出现在目标实例上，而当前实例上的 `key` 会被删除。

这个命令是一个原子操作，它在执行的时候会阻塞进行迁移的两个实例，直到以下任意结果发生：迁移成功，迁移失败，等到超时。

命令的内部实现是这样的：它在当前实例对给定 `key` 执行 *DUMP* 命令 ，将它序列化，然后传送到目标实例，目标实例再使用 *RESTORE* 对数据进行反序列化，并将反序列化所得的数据添加到数据库中；当前实例就像目标实例的客户端那样，只要看到 *RESTORE* 命令返回 `OK` ，它就会调用 *DEL* 删除自己数据库上的 `key` 。

`timeout` 参数以毫秒为格式，指定当前实例和目标实例进行沟通的最大间隔时间。这说明操作并不一定要在 `timeout` 毫秒内完成，只是说数据传送的时间不能超过这个 `timeout` 数。

MIGRATE 命令需要在给定的时间规定内完成 IO 操作。如果在传送数据时发生 IO 错误，或者达到了超时时间，那么命令会停止执行，并返回一个特殊的错误： `IOERR` 。

当 `IOERR` 出现时，有以下两种可能：

- `key` 可能存在于两个实例
- `key` 可能只存在于当前实例

唯一不可能发生的情况就是丢失 `key` ，因此，如果一个客户端执行 MIGRATE 命令，并且不幸遇上 `IOERR` 错误，那么这个客户端唯一要做的就是检查自己数据库上的 `key` 是否已经被正确地删除。

如果有其他错误发生，那么 MIGRATE 保证 `key` 只会出现在当前实例中。（当然，目标实例的给定数据库上可能有和 `key` 同名的键，不过这和 MIGRATE 命令没有关系）。

可选项：

- `COPY` ：不移除源实例上的 `key` 。
- `REPLACE` ：替换目标实例上已存在的 `key` 。

可用版本：

>= 2.6.0

时间复杂度：

这个命令在源实例上实际执行 *DUMP* 命令和 *DEL* 命令，在目标实例执行 *RESTORE* 命令，查看以上命令的文档可以看到详细的复杂度说明。 `key` 数据在两个实例之间传输的复杂度为 O(N) 。

返回值：

迁移成功时返回 `OK` ，否则返回相应的错误。

# 示例

先启动两个 Redis 实例，一个使用默认的 6379 端口，一个使用 7777 端口。

```
$ ./redis-server &
[1] 3557

...

$ ./redis-server --port 7777 &
[2] 3560

...
```

然后用客户端连上 6379 端口的实例，设置一个键，然后将它迁移到 7777 端口的实例上：

```
$ ./redis-cli

redis 127.0.0.1:6379> flushdb
OK

redis 127.0.0.1:6379> SET greeting "Hello from 6379 instance"
OK

redis 127.0.0.1:6379> MIGRATE 127.0.0.1 7777 greeting 0 1000
OK

redis 127.0.0.1:6379> EXISTS greeting                           # 迁移成功后 key 被删除
(integer) 0
```

使用另一个客户端，查看 7777 端口上的实例：

```
$ ./redis-cli -p 7777

redis 127.0.0.1:7777> GET greeting
"Hello from 6379 instance"
```

# MOVE

**MOVE key db**

将当前数据库的 `key` 移动到给定的数据库 `db` 当中。

如果当前数据库(源数据库)和给定数据库(目标数据库)有相同名字的给定 `key` ，或者 `key` 不存在于当前数据库，那么 `MOVE` 没有任何效果。

因此，也可以利用这一特性，将 MOVE 当作锁(locking)原语(primitive)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

移动成功返回 `1` ，失败则返回 `0` 。

```
# key 存在于当前数据库

redis> SELECT 0                             # redis默认使用数据库 0，为了清晰起见，这里再显式指定
OK

redis> SET song "secret base - Zone"
OK

redis> MOVE song 1                          # 将 song 移动到数据库 1
(integer) 1

redis> EXISTS song                          # song 已经被移走
(integer) 0

redis> SELECT 1                             # 使用数据库 1
OK

redis:1> EXISTS song                        # 证实 song 被移到了数据库 1 (注意命令提示符变成了"
(integer) 1

# 当 key 不存在的时候

redis:1> EXISTS fake_key
(integer) 0

redis:1> MOVE fake_key 0                    # 试图从数据库 1 移动一个不存在的 key 到数据库 0，失
(integer) 0

redis:1> select 0                           # 使用数据库0
OK

redis> EXISTS fake_key                      # 证实 fake_key 不存在
(integer) 0

# 当源数据库和目标数据库有相同的 key 时

redis> SELECT 0                             # 使用数据库0
OK
redis> SET favorite_fruit "banana"
OK

redis> SELECT 1                             # 使用数据库1
OK
redis:1> SET favorite_fruit "apple"
OK

redis:1> SELECT 0                           # 使用数据库0，并试图将 favorite_fruit 移动到数据库
OK

redis> MOVE favorite_fruit 1                # 因为两个数据库有相同的 key，MOVE 失败
(integer) 0

redis> GET favorite_fruit                   # 数据库 0 的 favorite_fruit 没变
"banana"

redis> SELECT 1
OK

redis:1> GET favorite_fruit                 # 数据库 1 的 favorite_fruit 也是
"apple"
```

# OBJECT

**OBJECT subcommand [arguments [arguments]]**

OBJECT 命令允许从内部察看给定 `key` 的 Redis 对象。

它通常用在除错(debugging)或者了解为了节省空间而对 `key` 使用特殊编码的情况。当将Redis用作缓存程序时，你也可以通过 OBJECT 命令中的信息，决定 `key` 的驱逐策略(eviction policies)。

OBJECT 命令有多个子命令：

- `OBJECT REFCOUNT <key>` 返回给定 `key` 引用所储存的值的次数。此命令主要用于除错。
- `OBJECT ENCODING <key>` 返回给定 `key` 锁储存的值所使用的内部表示(representation)。
- `OBJECT IDLETIME <key>` 返回给定 `key` 自储存以来的空闲时间(idle， 没有被读取也没有被写入)，以秒为单位。对象可以以多种方式编码：
- 字符串可以被编码为 `raw` (一般字符串)或 `int` (为了节约内存，Redis 会将字符串表示的 64 位有符号整数编码为整数来进行储存）。
- 列表可以被编码为 `ziplist` 或 `linkedlist` 。 `ziplist` 是为节约大小较小的列表空间而作的特殊表示。
- 集合可以被编码为 `intset` 或者 `hashtable` 。 `intset` 是只储存数字的小集合的特殊表示。
- 哈希表可以编码为 `zipmap` 或者 `hashtable` 。 `zipmap` 是小哈希表的特殊表示。
- 有序集合可以被编码为 `ziplist` 或者 `skiplist` 格式。 `ziplist` 用于表示小的有序集合，而 `skiplist` 则用于表示任何大小的有序集合。假如你做了什么让 Redis 没办法再使用节省空间的编码时(比如将一个只有 1 个元素的集合扩展为一个有 100 万个元素的集合)，特殊编码类型(specially encoded types)会自动转换成通用类型(general type)。

可用版本：

>= 2.2.3

时间复杂度：

O(1)

返回值：

`REFCOUNT` 和 `IDLETIME` 返回数字。 `ENCODING` 返回相应的编码类型。

```
redis> SET game "COD"           # 设置一个字符串
OK

redis> OBJECT REFCOUNT game     # 只有一个引用
(integer) 1

redis> OBJECT IDLETIME game     # 等待一阵。。。然后查看空闲时间
(integer) 90

redis> GET game                 # 提取game， 让它处于活跃(active)状态
"COD"

redis> OBJECT IDLETIME game     # 不再处于空闲状态
(integer) 0

redis> OBJECT ENCODING game     # 字符串的编码方式
"raw"

redis> SET big-number 23102930128301091820391092019203810281029831092  # 非常长的数字会被编
OK

redis> OBJECT ENCODING big-number
"raw"

redis> SET small-number 12345  # 而短的数字则会被编码为整数
OK

redis> OBJECT ENCODING small-number
"int"
```

# PERSIST

### PERSIST key

移除给定 `key` 的生存时间，将这个 `key` 从『易失的』(带生存时间 `key` )转换成『持久的』(一个不带生存时间、永不过期的 `key` )。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

当生存时间移除成功时，返回 `1` .如果 `key` 不存在或 `key` 没有设置生存时间，返回 `0` 。

```
redis> SET mykey "Hello"
OK

redis> EXPIRE mykey 10  # 为 key 设置生存时间
(integer) 1

redis> TTL mykey
(integer) 10

redis> PERSIST mykey    # 移除 key 的生存时间
(integer) 1

redis> TTL mykey
(integer) -1
```

# PEXPIRE

### PEXPIRE key milliseconds

这个命令和 *EXPIRE* 命令的作用类似，但是它以毫秒为单位设置 `key` 的生存时间，而不像 *EXPIRE* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1``key` 不存在或设置失败，返回 `0`

```
redis> SET mykey "Hello"
OK

redis> PEXPIRE mykey 1500
(integer) 1

redis> TTL mykey    # TTL 的返回值以秒为单位
(integer) 2

redis> PTTL mykey   # PTTL 可以给出准确的毫秒数
(integer) 1499
```

# PEXPIREAT

### PEXPIREAT key milliseconds-timestamp

这个命令和 *EXPIREAT* 命令类似，但它以毫秒为单位设置 `key` 的过期 unix 时间戳，而不是像 *EXPIREAT* 那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

如果生存时间设置成功，返回 `1` 。当 `key` 不存在或没办法设置生存时间时，返回 `0` 。(查看 *EXPIRE* 命令获取更多信息)

```
redis> SET mykey "Hello"
OK

redis> PEXPIREAT mykey 1555555555005
(integer) 1

redis> TTL mykey           # TTL 返回秒
(integer) 223157079

redis> PTTL mykey          # PTTL 返回毫秒
(integer) 223157079318
```

# PTTL

### PTTL key

这个命令类似于 *TTL* 命令，但它以毫秒为单位返回 `key` 的剩余生存时间，而不是像 *TTL* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `-2` 。当 `key` 存在但没有设置剩余生存时间时，返回 `-1` 。否则，以毫秒为单位，返回 `key` 的剩余生存时间。

Note

在 Redis 2.8 以前，当 `key` 不存在，或者 `key` 没有设置剩余生存时间时，命令都返回 `-1` 。

```
# 不存在的 key

redis> FLUSHDB
OK

redis> PTTL key
(integer) -2

# key 存在，但没有设置剩余生存时间

redis> SET key value
OK

redis> PTTL key
(integer) -1

# 有剩余生存时间的 key

redis> PEXPIRE key 10086
(integer) 1

redis> PTTL key
(integer) 6179
```

# RANDOMKEY

### RANDOMKEY

从当前数据库中随机返回(不删除)一个 `key` 。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当数据库不为空时，返回一个 `key` 。当数据库为空时，返回 `nil` 。

```
# 数据库不为空

redis> MSET fruit "apple" drink "beer" food "cookies"   # 设置多个 key
OK

redis> RANDOMKEY
"fruit"

redis> RANDOMKEY
"food"

redis> KEYS *    # 查看数据库内所有key，证明 RANDOMKEY 并不删除 key
1) "food"
2) "drink"
3) "fruit"

# 数据库为空

redis> FLUSHDB  # 删除当前数据库所有 key
OK

redis> RANDOMKEY
(nil)
```

# RENAME

### RENAME key newkey

将 `key` 改名为 `newkey` 。

当 `key` 和 `newkey` 相同，或者 `key` 不存在时，返回一个错误。

当 `newkey` 已经存在时， RENAME 命令将覆盖旧值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

改名成功时提示 `OK` ，失败时候返回一个错误。

```
# key 存在且 newkey 不存在

redis> SET message "hello world"
OK

redis> RENAME message greeting
OK

redis> EXISTS message                # message 不复存在
(integer) 0

redis> EXISTS greeting               # greeting 取而代之
(integer) 1

# 当 key 不存在时，返回错误

redis> RENAME fake_key never_exists
(error) ERR no such key

# newkey 已存在时， RENAME 会覆盖旧 newkey

redis> SET pc "lenovo"
OK

redis> SET personal_computer "dell"
OK

redis> RENAME pc personal_computer
OK

redis> GET pc
(nil)

redis:1> GET personal_computer      # 原来的值 dell 被覆盖了
"lenovo"
```

# RENAMENX

### RENAMENX key newkey

当且仅当 `newkey` 不存在时，将 `key` 改名为 `newkey` 。

当 `key` 不存在时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

修改成功时，返回 `1` 。如果 `newkey` 已经存在，返回 `0` 。

```
# newkey 不存在，改名成功

redis> SET player "MPlyaer"
OK

redis> EXISTS best_player
(integer) 0

redis> RENAMENX player best_player
(integer) 1

# newkey存在时，失败

redis> SET animal "bear"
OK

redis> SET favorite_animal "butterfly"
OK

redis> RENAMENX animal favorite_animal
(integer) 0

redis> get animal
"bear"

redis> get favorite_animal
"butterfly"
```

# RESTORE

**RESTORE key ttl serialized-value [REPLACE]**

反序列化给定的序列化值，并将它和给定的 `key` 关联。

参数 `ttl` 以毫秒为单位为 `key` 设置生存时间；如果 `ttl` 为 `0` ，那么不设置生存时间。

RESTORE 在执行反序列化之前会先对序列化值的 RDB 版本和数据校验和进行检查，如果 RDB 版本不相同或者数据不完整的话，那么 RESTORE 会拒绝进行反序列化，并返回一个错误。

如果键 `key` 已经存在， 并且给定了 `REPLACE` 选项， 那么使用反序列化得出的值来代替键 `key` 原有的值； 相反地， 如果键 `key` 已经存在， 但是没有给定 `REPLACE` 选项， 那么命令返回一个错误。

更多信息可以参考 *DUMP* 命令。

可用版本：

>= 2.6.0

时间复杂度：

查找给定键的复杂度为 O(1) ，对键进行反序列化的复杂度为 O(N*M*) ，其中 *N* 是构成 *key* 的 *Redis* 对象的数量，而 *M* 则是这些对象的平均大小。有序集合*(sorted set)*的反序列化复杂度为 *O(N*M*log(N)) ，因为有序集合每次插入的复杂度为 O(log(N)) 。如果反序列化的对象是比较小的字符串，那么复杂度为 O(1) 。

返回值：

如果反序列化成功那么返回 `OK` ，否则返回一个错误。

```
# 创建一个键，作为 DUMP 命令的输入

redis> SET greeting "hello, dumping world!"
OK

redis> DUMP greeting
"\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1\xde"

# 将序列化数据 RESTORE 到另一个键上面

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1
OK

redis> GET greeting-again
"hello, dumping world!"

# 在没有给定 REPLACE 选项的情况下，再次尝试反序列化到同一个键，失败

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1
(error) ERR Target key name is busy.

# 给定 REPLACE 选项，对同一个键进行反序列化成功

redis> RESTORE greeting-again 0 "\x00\x15hello, dumping world!\x06\x00E\xa0Z\x82\xd8r\xc1
OK

# 尝试使用无效的值进行反序列化，出错

redis> RESTORE fake-message 0 "hello moto moto blah blah"
(error) ERR DUMP payload version or checksum are wrong
```

# SORT

**SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC | DESC] [ALPHA] [STORE destination]**

返回或保存给定列表、集合、有序集合 `key` 中经过排序的元素。

排序默认以数字作为对象，值被解释为双精度浮点数，然后进行比较。

## 一般 SORT 用法

最简单的 SORT 使用方法是 `SORT key` 和 `SORT key DESC` ：

- `SORT key` 返回键值从小到大排序的结果。
- `SORT key DESC` 返回键值从大到小排序的结果。

假设 `today_cost` 列表保存了今日的开销金额， 那么可以用 SORT 命令对它进行排序：

```
# 开销金额列表

redis> LPUSH today_cost 30 1.5 10 8
(integer) 4

# 排序

redis> SORT today_cost
1) "1.5"
2) "8"
3) "10"
4) "30"

# 逆序排序

redis 127.0.0.1:6379> SORT today_cost DESC
1) "30"
2) "10"
3) "8"
4) "1.5"
```

## 使用 ALPHA 修饰符对字符串进行排序

因为 SORT 命令默认排序对象为数字， 当需要对字符串进行排序时， 需要显式地在 SORT 命令之后添加 `ALPHA` 修饰符：

```
# 网址

redis> LPUSH website "www.reddit.com"
(integer) 1

redis> LPUSH website "www.slashdot.com"
(integer) 2

redis> LPUSH website "www.infoq.com"
(integer) 3

# 默认（按数字）排序

redis> SORT website
1) "www.infoq.com"
2) "www.slashdot.com"
3) "www.reddit.com"

# 按字符排序

redis> SORT website ALPHA
1) "www.infoq.com"
2) "www.reddit.com"
3) "www.slashdot.com"
```

如果系统正确地设置了 `LC_COLLATE` 环境变量的话，Redis能识别 `UTF-8` 编码。

# 使用 LIMIT 修饰符限制返回结果

排序之后返回元素的数量可以通过 `LIMIT` 修饰符进行限制， 修饰符接受 `offset` 和 `count` 两个参数：

- `offset` 指定要跳过的元素数量。
- `count` 指定跳过 `offset` 个指定的元素之后，要返回多少个对象。

以下例子返回排序结果的前 5 个对象( `offset` 为 `0` 表示没有元素被跳过)。

```
# 添加测试数据，列表值为 1 指 10

redis 127.0.0.1:6379> RPUSH rank 1 3 5 7 9
(integer) 5

redis 127.0.0.1:6379> RPUSH rank 2 4 6 8 10
(integer) 10

# 返回列表中最小的 5 个值

redis 127.0.0.1:6379> SORT rank LIMIT 0 5
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
```

可以组合使用多个修饰符。以下例子返回从大到小排序的前 5 个对象。

```
redis 127.0.0.1:6379> SORT rank LIMIT 0 5 DESC
1) "10"
2) "9"
3) "8"
4) "7"
5) "6"
```

# 使用外部 key 进行排序

可以使用外部 `key` 的数据作为权重，代替默认的直接对比键值的方式来进行排序。

假设现在有用户数据如下：

| uid | user*name*{uid} | user*level*{uid} |
| --- | --- | --- |
| 1 | admin | 9999 |
| 2 | jack | 10 |
| 3 | peter | 25 |
| 4 | mary | 70 |

以下代码将数据输入到 Redis 中：

```
# admin

redis 127.0.0.1:6379> LPUSH uid 1
(integer) 1

redis 127.0.0.1:6379> SET user_name_1 admin
OK

redis 127.0.0.1:6379> SET user_level_1 9999
OK

# jack

redis 127.0.0.1:6379> LPUSH uid 2
(integer) 2

redis 127.0.0.1:6379> SET user_name_2 jack
OK

redis 127.0.0.1:6379> SET user_level_2 10
OK

# peter

redis 127.0.0.1:6379> LPUSH uid 3
(integer) 3

redis 127.0.0.1:6379> SET user_name_3 peter
OK

redis 127.0.0.1:6379> SET user_level_3 25
OK

# mary

redis 127.0.0.1:6379> LPUSH uid 4
(integer) 4

redis 127.0.0.1:6379> SET user_name_4 mary
OK

redis 127.0.0.1:6379> SET user_level_4 70
OK
```

## BY 选项

默认情况下， `SORT uid` 直接按 `uid` 中的值排序：

```
redis 127.0.0.1:6379> SORT uid
1) "1"      # admin
2) "2"      # jack
3) "3"      # peter
4) "4"      # mary
```

通过使用 `BY` 选项，可以让 `uid` 按其他键的元素来排序。

比如说， 以下代码让 `uid` 键按照 `user_level_{uid}` 的大小来排序：

```
redis 127.0.0.1:6379> SORT uid BY user_level_*
1) "2"      # jack , level = 10
2) "3"      # peter, level = 25
3) "4"      # mary, level = 70
4) "1"      # admin, level = 9999
```

`user_level_*` 是一个占位符， 它先取出 `uid` 中的值， 然后再用这个值来查找相应的键。

比如在对 `uid` 列表进行排序时， 程序就会先取出 `uid` 的值 `1` 、 `2` 、 `3` 、 `4` ， 然后使用 `user_level_1` 、 `user_level_2` 、 `user_level_3` 和 `user_level_4` 的值作为排序 `uid` 的权重。

## GET 选项

使用 `GET` 选项， 可以根据排序的结果来取出相应的键值。

比如说， 以下代码先排序 `uid` ， 再取出键 `user_name_{uid}` 的值：

```
redis 127.0.0.1:6379> SORT uid GET user_name_*
1) "admin"
2) "jack"
3) "peter"
4) "mary"
```

## 组合使用 BY 和 GET

通过组合使用 `BY` 和 `GET` ， 可以让排序结果以更直观的方式显示出来。

比如说， 以下代码先按 `user_level_{uid}` 来排序 `uid` 列表， 再取出相应的 `user_name_{uid}` 的值：

```
redis 127.0.0.1:6379> SORT uid BY user_level_* GET user_name_*
1) "jack"       # level = 10
2) "peter"      # level = 25
3) "mary"       # level = 70
4) "admin"      # level = 9999
```

现在的排序结果要比只使用 `SORT uid BY user_level_*` 要直观得多。

## 获取多个外部键

可以同时使用多个 `GET` 选项， 获取多个外部键的值。

以下代码就按 `uid` 分别获取 `user_level_{uid}` 和 `user_name_{uid}` ：

```
redis 127.0.0.1:6379> SORT uid GET user_level_* GET user_name_*
1) "9999"       # level
2) "admin"      # name
3) "10"
4) "jack"
5) "25"
6) "peter"
7) "70"
8) "mary"
```

`GET` 有一个额外的参数规则，那就是 —— 可以用 `#` 获取被排序键的值。

以下代码就将 `uid` 的值、及其相应的 `user_level_*` 和 `user_name_*` 都返回为结果：

```
redis 127.0.0.1:6379> SORT uid GET # GET user_level_* GET user_name_*
1) "1"          # uid
2) "9999"       # level
3) "admin"      # name
4) "2"
5) "10"
6) "jack"
7) "3"
8) "25"
9) "peter"
10) "4"
11) "70"
12) "mary"
```

## 获取外部键，但不进行排序

通过将一个不存在的键作为参数传给 `BY` 选项， 可以让 `SORT` 跳过排序操作， 直接返回结果：

```
redis 127.0.0.1:6379> SORT uid BY not-exists-key
1) "4"
2) "3"
3) "2"
4) "1"
```

这种用法在单独使用时，没什么实际用处。

不过，通过将这种用法和 `GET` 选项配合， 就可以在不排序的情况下， 获取多个外部键， 相当于执行一个整合的获取操作（类似于 SQL 数据库的 `join` 关键字）。

以下代码演示了，如何在不引起排序的情况下，使用 `SORT` 、 `BY` 和 `GET` 获取多个外部键：

```
redis 127.0.0.1:6379> SORT uid BY not-exists-key GET # GET user_level_* GET user_name_*
1) "4"      # id
2) "70"     # level
3) "mary"   # name
4) "3"
5) "25"
6) "peter"
7) "2"
8) "10"
9) "jack"
10) "1"
11) "9999"
12) "admin"
```

## 将哈希表作为 GET 或 BY 的参数

除了可以将字符串键之外， 哈希表也可以作为 `GET` 或 `BY` 选项的参数来使用。

比如说，对于前面给出的用户信息表：

| uid | user*name*{uid} | user*level*{uid} |
| --- | --- | --- |
| 1 | admin | 9999 |
| 2 | jack | 10 |
| 3 | peter | 25 |
| 4 | mary | 70 |

我们可以不将用户的名字和级别保存在 `user_name_{uid}` 和 `user_level_{uid}` 两个字符串键中， 而是用一个带有 `name` 域和 `level` 域的哈希表 `user_info_{uid}` 来保存用户的名字和级别信息：

```
redis 127.0.0.1:6379> HMSET user_info_1 name admin level 9999
OK

redis 127.0.0.1:6379> HMSET user_info_2 name jack level 10
OK

redis 127.0.0.1:6379> HMSET user_info_3 name peter level 25
OK

redis 127.0.0.1:6379> HMSET user_info_4 name mary level 70
OK
```

之后， `BY` 和 `GET` 选项都可以用 `key->field` 的格式来获取哈希表中的域的值， 其中 `key` 表示哈希表键， 而 `field` 则表示哈希表的域：

```
redis 127.0.0.1:6379> SORT uid BY user_info_*->level
1) "2"
2) "3"
3) "4"
4) "1"

redis 127.0.0.1:6379> SORT uid BY user_info_*->level GET user_info_*->name
1) "jack"
2) "peter"
3) "mary"
4) "admin"
```

# 保存排序结果

默认情况下， SORT 操作只是简单地返回排序结果，并不进行任何保存操作。

通过给 `STORE` 选项指定一个 `key` 参数，可以将排序结果保存到给定的键上。

如果被指定的 `key` 已存在，那么原有的值将被排序结果覆盖。

```
# 测试数据

redis 127.0.0.1:6379> RPUSH numbers 1 3 5 7 9
(integer) 5

redis 127.0.0.1:6379> RPUSH numbers 2 4 6 8 10
(integer) 10

redis 127.0.0.1:6379> LRANGE numbers 0 -1
1) "1"
2) "3"
3) "5"
4) "7"
5) "9"
6) "2"
7) "4"
8) "6"
9) "8"
10) "10"

redis 127.0.0.1:6379> SORT numbers STORE sorted-numbers
(integer) 10

# 排序后的结果

redis 127.0.0.1:6379> LRANGE sorted-numbers 0 -1
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
10) "10"
```

可以通过将 SORT 命令的执行结果保存，并用 *EXPIRE* 为结果设置生存时间，以此来产生一个 SORT 操作的结果缓存。

这样就可以避免对 SORT 操作的频繁调用：只有当结果集过期时，才需要再调用一次 SORT 操作。

另外，为了正确实现这一用法，你可能需要加锁以避免多个客户端同时进行缓存重建(也就是多个客户端，同一时间进行 SORT 操作，并保存为结果集)，具体参见 *SETNX* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(N+M*log(M))， `N` 为要排序的列表或集合内的元素数量， `M` 为要返回的元素数量。如果只是使用 SORT 命令的 `GET` 选项获取数据而没有进行排序，时间复杂度 O(N)。

返回值：

没有使用 `STORE` 参数，返回列表形式的排序结果。使用 `STORE` 参数，返回排序结果的元素数量。

# TTL

**TTL key**

以秒为单位，返回给定 `key` 的剩余生存时间(TTL, time to live)。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `-2` 。当 `key` 存在但没有设置剩余生存时间时，返回 `-1` 。否则，以秒为单位，返回 `key` 的剩余生存时间。

Note

在 Redis 2.8 以前，当 `key` 不存在，或者 `key` 没有设置剩余生存时间时，命令都返回 `-1` 。

```
# 不存在的 key

redis> FLUSHDB
OK

redis> TTL key
(integer) -2

# key 存在，但没有设置剩余生存时间

redis> SET key value
OK

redis> TTL key
(integer) -1

# 有剩余生存时间的 key

redis> EXPIRE key 10086
(integer) 1

redis> TTL key
(integer) 10084
```

# TYPE

### TYPE key

返回 `key` 所储存的值的类型。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`none` (key不存在) `string` (字符串) `list` (列表) `set` (集合) `zset` (有序集) `hash` (哈希表)

```
# 字符串

redis> SET weather "sunny"
OK

redis> TYPE weather
string

# 列表

redis> LPUSH book_list "programming in scala"
(integer) 1

redis> TYPE book_list
list

# 集合

redis> SADD pat "dog"
(integer) 1

redis> TYPE pat
set
```

# SCAN

**SCAN cursor [MATCH pattern] [COUNT count]**

*SCAN* 命令及其相关的 *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令都用于增量地迭代（incrementally iterate）一集元素（a collection of elements）：

- *SCAN* 命令用于迭代当前数据库中的数据库键。
- *SSCAN* 命令用于迭代集合键中的元素。
- *HSCAN* 命令用于迭代哈希键中的键值对。
- *ZSCAN* 命令用于迭代有序集合中的元素（包括元素成员和元素分值）。

以上列出的四个命令都支持增量式迭代， 它们每次执行都只会返回少量元素， 所以这些命令可以用于生产环境， 而不会出现像 *KEYS* 命令、 *SMEMBERS* 命令带来的问题 —— 当 *KEYS* 命令被用于处理一个大的数据库时， 又或者 *SMEMBERS* 命令被用于处理一个大的集合键时， 它们可能会阻塞服务器达数秒之久。

不过， 增量式迭代命令也不是没有缺点的： 举个例子， 使用 *SMEMBERS* 命令可以返回集合键当前包含的所有元素， 但是对于 *SCAN* 这类增量式迭代命令来说， 因为在对键进行增量式迭代的过程中， 键可能会被修改， 所以增量式迭代命令只能对被返回的元素提供有限的保证 （offer limited guarantees about the returned elements）。

因为 *SCAN* 、 *SSCAN* 、 *HSCAN* 和 *ZSCAN* 四个命令的工作方式都非常相似， 所以这个文档会一并介绍这四个命令， 但是要记住：

- *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令的第一个参数总是一个数据库键。
- 而 *SCAN* 命令则不需要在第一个参数提供任何数据库键 —— 因为它迭代的是当前数据库中的所有数据库键。

## SCAN 命令的基本用法

*SCAN* 命令是一个基于游标的迭代器（cursor based iterator）： *SCAN* 命令每次被调用之后， 都会向用户返回一个新的游标， 用户在下次迭代时需要使用这个新游标作为 *SCAN* 命令的游标参数， 以此来延续之前的迭代过程。

当 *SCAN* 命令的游标参数被设置为 `0` 时， 服务器将开始一次新的迭代， 而当服务器向用户返回值为 `0` 的游标时， 表示迭代已结束。

以下是一个 *SCAN* 命令的迭代过程示例：

```
redis 127.0.0.1:6379> scan 0
1) "17"
2)  1) "key:12"
    2) "key:8"
    3) "key:4"
    4) "key:14"
    5) "key:16"
    6) "key:17"
    7) "key:15"
    8) "key:10"
    9) "key:3"
    10) "key:7"
    11) "key:1"

redis 127.0.0.1:6379> scan 17
1) "0"
2) 1) "key:5"
   2) "key:18"
   3) "key:0"
   4) "key:2"
   5) "key:19"
   6) "key:13"
   7) "key:6"
   8) "key:9"
   9) "key:11"
```

在上面这个例子中， 第一次迭代使用 `0` 作为游标， 表示开始一次新的迭代。

第二次迭代使用的是第一次迭代时返回的游标， 也即是命令回复第一个元素的值 —— `17` 。

从上面的示例可以看到， *SCAN* 命令的回复是一个包含两个元素的数组， 第一个数组元素是用于进行下一次迭代的新游标， 而第二个数组元素则是一个数组， 这个数组中包含了所有被迭代的元素。

在第二次调用 *SCAN* 命令时， 命令返回了游标 `0` ， 这表示迭代已经结束， 整个数据集（collection）已经被完整遍历过了。

以 `0` 作为游标开始一次新的迭代， 一直调用 *SCAN* 命令， 直到命令返回游标 `0` ， 我们称这个过程为一次完整遍历（full iteration）。

# SCAN 命令的保证（guarantees）

*SCAN* 命令， 以及其他增量式迭代命令， 在进行完整遍历的情况下可以为用户带来以下保证 ： 从完整遍历开始直到完整遍历结束期间， 一直存在于数据集内的所有元素都会被完整遍历返回； 这意味着， 如果有一个元素， 它从遍历开始直到遍历结束期间都存在于被遍历的数据集当中， 那么 *SCAN* 命令总会在某次迭代中将这个元素返回给用户。

然而因为增量式命令仅仅使用游标来记录迭代状态， 所以这些命令带有以下缺点 ：

- 同一个元素可能会被返回多次。 处理重复元素的工作交由应用程序负责， 比如说， 可以考虑将迭代返回的元素仅仅用于可以安全地重复执行多次的操作上。
- 如果一个元素是在迭代过程中被添加到数据集的， 又或者是在迭代过程中从数据集中被删除的， 那么这个元素可能会被返回， 也可能不会， 这是未定义的（undefined）。

## SCAN 命令每次执行返回的元素数量

增量式迭代命令并不保证每次执行都返回某个给定数量的元素。

增量式命令甚至可能会返回零个元素， 但只要命令返回的游标不是 `0` ， 应用程序就不应该将迭代视作结束。

不过命令返回的元素数量总是符合一定规则的， 在实际中：

- 对于一个大数据集来说， 增量式迭代命令每次最多可能会返回数十个元素；
- 而对于一个足够小的数据集来说， 如果这个数据集的底层表示为编码数据结构（encoded data structure，适用于是小集合键、小哈希键和小有序集合键）， 那么增量迭代命令将在一次调用中返回数据集中的所有元素。

最后， 用户可以通过增量式迭代命令提供的 `COUNT` 选项来指定每次迭代返回元素的最大值。

## COUNT 选项

虽然增量式迭代命令不保证每次迭代所返回的元素数量， 但我们可以使用 `COUNT` 选项， 对命令的行为进行一定程度上的调整。

基本上， `COUNT` 选项的作用就是让用户告知迭代命令， 在每次迭代中应该从数据集里返回多少元素。

虽然 `COUNT` 选项只是对增量式迭代命令的一种提示（hint）， 但是在大多数情况下， 这种提示都是有效的。

- `COUNT` 参数的默认值为 `10` 。
- 在迭代一个足够大的、由哈希表实现的数据库、集合键、哈希键或者有序集合键时， 如果用户没有使用 `MATCH` 选项， 那么命令返回的元素数量通常和 `COUNT` 选项指定的一样， 或者比 `COUNT` 选项指定的数量稍多一些。
- 在迭代一个编码为整数集合（intset，一个只由整数值构成的小集合）、 或者编码为压缩列表（ziplist，由不同值构成的一个小哈希或者一个小有序集合）时， 增量式迭代命令通常会无视 `COUNT` 选项指定的值， 在第一次迭代就将数据集包含的所有元素都返回给用户。

Note

并非每次迭代都要使用相同的 `COUNT` 值。

用户可以在每次迭代中按自己的需要随意改变 `COUNT` 值， 只要记得将上次迭代返回的游标用到下次迭代里面就可以了。

# MATCH 选项

和 *KEYS* 命令一样， 增量式迭代命令也可以通过提供一个 glob 风格的模式参数， 让命令只返回和给定模式相匹配的元素， 这一点可以通过在执行增量式迭代命令时， 通过给定 `MATCH <pattern>` 参数来实现。

以下是一个使用 `MATCH` 选项进行迭代的示例：

```
redis 127.0.0.1:6379> sadd myset 1 2 3 foo foobar feelsgood
(integer) 6

redis 127.0.0.1:6379> sscan myset 0 match f*
1) "0"
2) 1) "foo"
   2) "feelsgood"
   3) "foobar"
```

需要注意的是， 对元素的模式匹配工作是在命令从数据集中取出元素之后， 向客户端返回元素之前的这段时间内进行的， 所以如果被迭代的数据集中只有少量元素和模式相匹配， 那么迭代命令或许会在多次执行中都不返回任何元素。

以下是这种情况的一个例子：

```
redis 127.0.0.1:6379> scan 0 MATCH *11*
1) "288"
2) 1) "key:911"

redis 127.0.0.1:6379> scan 288 MATCH *11*
1) "224"
2) (empty list or set)

redis 127.0.0.1:6379> scan 224 MATCH *11*
1) "80"
2) (empty list or set)

redis 127.0.0.1:6379> scan 80 MATCH *11*
1) "176"
2) (empty list or set)

redis 127.0.0.1:6379> scan 176 MATCH *11* COUNT 1000
1) "0"
2)  1) "key:611"
    2) "key:711"
    3) "key:118"
    4) "key:117"
    5) "key:311"
    6) "key:112"
    7) "key:111"
    8) "key:110"
    9) "key:113"
   10) "key:211"
   11) "key:411"
   12) "key:115"
   13) "key:116"
   14) "key:114"
   15) "key:119"
   16) "key:811"
   17) "key:511"
   18) "key:11"
```

如你所见， 以上的大部分迭代都不返回任何元素。

在最后一次迭代， 我们通过将 `COUNT` 选项的参数设置为 `1000` ， 强制命令为本次迭代扫描更多元素， 从而使得命令返回的元素也变多了。

# 并发执行多个迭代

在同一时间， 可以有任意多个客户端对同一数据集进行迭代， 客户端每次执行迭代都需要传入一个游标， 并在迭代执行之后获得一个新的游标， 而这个游标就包含了迭代的所有状态， 因此， 服务器无须为迭代记录任何状态。

# 中途停止迭代

因为迭代的所有状态都保存在游标里面， 而服务器无须为迭代保存任何状态， 所以客户端可以在中途停止一个迭代， 而无须对服务器进行任何通知。

即使有任意数量的迭代在中途停止， 也不会产生任何问题。

# 使用错误的游标进行增量式迭代

使用间断的（broken）、负数、超出范围或者其他非正常的游标来执行增量式迭代并不会造成服务器崩溃， 但可能会让命令产生未定义的行为。

未定义行为指的是， 增量式命令对返回值所做的保证可能会不再为真。

只有两种游标是合法的：

1. 在开始一个新的迭代时， 游标必须为 `0` 。
2. 增量式迭代命令在执行之后返回的， 用于延续（continue）迭代过程的游标。

# 迭代终结的保证

增量式迭代命令所使用的算法只保证在数据集的大小有界（bounded）的情况下， 迭代才会停止， 换句话说， 如果被迭代数据集的大小不断地增长的话， 增量式迭代命令可能永远也无法完成一次完整迭代。

从直觉上可以看出， 当一个数据集不断地变大时， 想要访问这个数据集中的所有元素就需要做越来越多的工作， 能否结束一个迭代取决于用户执行迭代的速度是否比数据集增长的速度更快。

可用版本：

> >= 2.8.0

时间复杂度：

> 增量式迭代命令每次执行的复杂度为 O(1)， 对数据集进行一次完整迭代的复杂度为 O(N) ， 其中 N 为数据集中的元素数量。

返回值：

> *SCAN* 命令、 *SSCAN* 命令、 *HSCAN* 命令和 *ZSCAN* 命令都返回一个包含两个元素的 multi-bulk 回复： 回复的第一个元素是字符串表示的无符号 64 位整数（游标）， 回复的第二个元素是另一个 multi-bulk 回复， 这个 multi-bulk 回复包含了本次被迭代的元素。 > > *SCAN* 命令返回的每个元素都是一个数据库键。 > > *SSCAN* 命令返回的每个元素都是一个集合成员。 > > *HSCAN* 命令返回的每个元素都是一个键值对，一个键值对由一个键和一个值组成。 > > *ZSCAN* 命令返回的每个元素都是一个有序集合元素，一个有序集合元素由一个成员（member）和一个分值（score）组成。

# String（字符串）

# APPEND

### APPEND key value

如果 `key` 已经存在并且是一个字符串， APPEND 命令将 `value` 追加到 `key` 原来的值的末尾。

如果 `key` 不存在， APPEND 就简单地将给定 `key` 设为 `value` ，就像执行 `SET key value` 一样。

可用版本：

>= 2.0.0

时间复杂度：

平摊O(1)

返回值：

追加 `value` 之后， `key` 中字符串的长度。

```
# 对不存在的 key 执行 APPEND

redis> EXISTS myphone               # 确保 myphone 不存在
(integer) 0

redis> APPEND myphone "nokia"       # 对不存在的 key 进行 APPEND ，等同于 SET myphone "nokia"
(integer) 5                         # 字符长度

# 对已存在的字符串进行 APPEND

redis> APPEND myphone " - 1110"     # 长度从 5 个字符增加到 12 个字符
(integer) 12

redis> GET myphone
"nokia - 1110"
```

# 模式：时间序列(Time series)

APPEND 可以为一系列定长(fixed-size)数据(sample)提供一种紧凑的表示方式，通常称之为时间序列。

每当一个新数据到达的时候，执行以下命令：

```
APPEND timeseries "fixed-size sample"
```

然后可以通过以下的方式访问时间序列的各项属性：

- *STRLEN* 给出时间序列中数据的数量
- *GETRANGE* 可以用于随机访问。只要有相关的时间信息的话，我们就可以在 Redis 2.6 中使用 Lua 脚本和 *GETRANGE* 命令实现二分查找。
- *SETRANGE* 可以用于覆盖或修改已存在的的时间序列。

这个模式的唯一缺陷是我们只能增长时间序列，而不能对时间序列进行缩短，因为 Redis 目前还没有对字符串进行修剪(tirm)的命令，但是，不管怎么说，这个模式的储存方式还是可以节省下大量的空间。

Note

可以考虑使用 UNIX 时间戳作为时间序列的键名，这样一来，可以避免单个 `key` 因为保存过大的时间序列而占用大量内存，另一方面，也可以节省下大量命名空间。

下面是一个时间序列的例子：

```
redis> APPEND ts "0043"
(integer) 4

redis> APPEND ts "0035"
(integer) 8

redis> GETRANGE ts 0 3
"0043"

redis> GETRANGE ts 4 7
"0035"
```

# BITCOUNT

**BITCOUNT key [start] [end]**

计算给定字符串中，被设置为 `1` 的比特位的数量。

一般情况下，给定的整个字符串都会被进行计数，通过指定额外的 `start` 或 `end` 参数，可以让计数只在特定的位上进行。

`start` 和 `end` 参数的设置和 *GETRANGE* 命令类似，都可以使用负数值： 比如 `-1` 表示最后一个字节， `-2` 表示倒数第二个字节，以此类推。

不存在的 `key` 被当成是空字符串来处理，因此对一个不存在的 `key` 进行 `BITCOUNT` 操作，结果为 `0` 。

可用版本：

>= 2.6.0

时间复杂度：

O(N)

返回值：

被设置为 `1` 的位的数量。

```
redis> BITCOUNT bits
(integer) 0

redis> SETBIT bits 0 1          # 0001
(integer) 0

redis> BITCOUNT bits
(integer) 1

redis> SETBIT bits 3 1          # 1001
(integer) 0

redis> BITCOUNT bits
(integer) 2
```

## 模式：使用 bitmap 实现用户上线次数统计

Bitmap 对于一些特定类型的计算非常有效。

假设现在我们希望记录自己网站上的用户的上线频率，比如说，计算用户 A 上线了多少天，用户 B 上线了多少天，诸如此类，以此作为数据，从而决定让哪些用户参加 beta 测试等活动 —— 这个模式可以使用 *SETBIT* 和 *BITCOUNT* 来实现。

比如说，每当用户在某一天上线的时候，我们就使用 *SETBIT* ，以用户名作为 `key` ，将那天所代表的网站的上线日作为 `offset` 参数，并将这个 `offset` 上的为设置为 `1` 。

举个例子，如果今天是网站上线的第 100 天，而用户 peter 在今天阅览过网站，那么执行命令 `SETBIT peter 100 1` ；如果明天 peter 也继续阅览网站，那么执行命令 `SETBIT peter 101 1` ，以此类推。

当要计算 peter 总共以来的上线次数时，就使用 *BITCOUNT* 命令：执行 `BITCOUNT peter` ，得出的结果就是 peter 上线的总天数。

更详细的实现可以参考博文(墙外) Fast, easy, realtime metrics using Redis bitmaps 。

# 性能

前面的上线次数统计例子，即使运行 10 年，占用的空间也只是每个用户 10*365 比特位(bit)，也即是每个用户 456 字节。对于这种大小的数据来说， *BITCOUNT* 的处理速度就像 *GET* 和 *INCR* 这种 O(1) 复杂度的操作一样快。

如果你的 bitmap 数据非常大，那么可以考虑使用以下两种方法：

1. 将一个大的 bitmap 分散到不同的 key 中，作为小的 bitmap 来处理。使用 Lua 脚本可以很方便地完成这一工作。
2. 使用 *BITCOUNT* 的 `start` 和 `end` 参数，每次只对所需的部分位进行计算，将位的累积工作(accumulating)放到客户端进行，并且对结果进行缓存 (caching)。

# BITOP

### BITOP operation destkey key [key ...]

对一个或多个保存二进制位的字符串 `key` 进行位元操作，并将结果保存到 `destkey` 上。

`operation` 可以是 `AND` 、 `OR` 、 `NOT` 、 `XOR` 这四种操作中的任意一种：

- `BITOP AND destkey key [key ...]` ，对一个或多个 `key` 求逻辑并，并将结果保存到 `destkey` 。
- `BITOP OR destkey key [key ...]` ，对一个或多个 `key` 求逻辑或，并将结果保存到 `destkey` 。
- `BITOP XOR destkey key [key ...]` ，对一个或多个 `key` 求逻辑异或，并将结果保存到 `destkey` 。
- `BITOP NOT destkey key` ，对给定 `key` 求逻辑非，并将结果保存到 `destkey` 。

除了 `NOT` 操作之外，其他操作都可以接受一个或多个 `key` 作为输入。

处理不同长度的字符串

当 BITOP 处理不同长度的字符串时，较短的那个字符串所缺少的部分会被看作 `0` 。

空的 `key` 也被看作是包含 `0` 的字符串序列。

可用版本：

>= 2.6.0

时间复杂度：

O(N)

返回值：

保存到 `destkey` 的字符串的长度，和输入 `key` 中最长的字符串长度相等。

Note

BITOP 的复杂度为 O(N) ，当处理大型矩阵(matrix)或者进行大数据量的统计时，最好将任务指派到附属节点(slave)进行，避免阻塞主节点。

```
redis> SETBIT bits-1 0 1        # bits-1 = 1001
(integer) 0

redis> SETBIT bits-1 3 1
(integer) 0

redis> SETBIT bits-2 0 1        # bits-2 = 1011
(integer) 0

redis> SETBIT bits-2 1 1
(integer) 0

redis> SETBIT bits-2 3 1
(integer) 0

redis> BITOP AND and-result bits-1 bits-2
(integer) 1

redis> GETBIT and-result 0      # and-result = 1001
(integer) 1

redis> GETBIT and-result 1
(integer) 0

redis> GETBIT and-result 2
(integer) 0

redis> GETBIT and-result 3
(integer) 1
```

# DECR

### DECR key

将 `key` 中储存的数字值减一。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 DECR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，请参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 DECR 命令之后 `key` 的值。

```
# 对存在的数字值 key 进行 DECR

redis> SET failure_times 10
OK

redis> DECR failure_times
(integer) 9

# 对不存在的 key 值进行 DECR

redis> EXISTS count
(integer) 0

redis> DECR count
(integer) -1

# 对存在但不是数值的 key 进行 DECR

redis> SET company YOUR_CODE_SUCKS.LLC
OK

redis> DECR company
(error) ERR value is not an integer or out of range
```

# DECRBY

### DECRBY key decrement

将 `key` 所储存的值减去减量 `decrement` 。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 DECRBY 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于更多递增(increment) / 递减(decrement)操作的更多信息，请参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

减去 `decrement` 之后， `key` 的值。

```
# 对已存在的 key 进行 DECRBY

redis> SET count 100
OK

redis> DECRBY count 20
(integer) 80

# 对不存在的 key 进行DECRBY

redis> EXISTS pages
(integer) 0

redis> DECRBY pages 10
(integer) -10
```

# GET

### GET key

返回 `key` 所关联的字符串值。

如果 `key` 不存在那么返回特殊值 `nil` 。

假如 `key` 储存的值不是字符串类型，返回一个错误，因为 GET 只能用于处理字符串值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 不存在时，返回 `nil` ，否则，返回 `key` 的值。如果 `key` 不是字符串类型，那么返回一个错误。

```
# 对不存在的 key 或字符串类型 key 进行 GET

redis> GET db
(nil)

redis> SET db redis
OK

redis> GET db
"redis"

# 对不是字符串类型的 key 进行 GET

redis> DEL db
(integer) 1

redis> LPUSH db redis mongodb mysql
(integer) 3

redis> GET db
(error) ERR Operation against a key holding the wrong kind of value
```

# GETBIT

### GETBIT key offset

对 `key` 所储存的字符串值，获取指定偏移量上的位(bit)。

当 `offset` 比字符串值的长度大，或者 `key` 不存在时，返回 `0` 。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

字符串值指定偏移量上的位(bit)。

```
# 对不存在的 key 或者不存在的 offset 进行 GETBIT， 返回 0

redis> EXISTS bit
(integer) 0

redis> GETBIT bit 10086
(integer) 0

# 对已存在的 offset 进行 GETBIT

redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1
```

# GETRANGE

### GETRANGE key start end

返回 `key` 中字符串值的子字符串，字符串的截取范围由 `start` 和 `end` 两个偏移量决定(包括 `start` 和 `end` 在内)。

负数偏移量表示从字符串最后开始计数， `-1` 表示最后一个字符， `-2` 表示倒数第二个，以此类推。

GETRANGE 通过保证子字符串的值域(range)不超过实际字符串的值域来处理超出范围的值域请求。

Note

在 <= 2.0 的版本里，GETRANGE 被叫作 SUBSTR。

可用版本：

>= 2.4.0

时间复杂度：

O(N)， `N` 为要返回的字符串的长度。复杂度最终由字符串的返回值长度决定，但因为从已有字符串中取出子字符串的操作非常廉价(cheap)，所以对于长度不大的字符串，该操作的复杂度也可看作O(1)。

返回值：

截取得出的子字符串。

```
redis> SET greeting "hello, my friend"
OK

redis> GETRANGE greeting 0 4          # 返回索引0-4的字符，包括4。
"hello"

redis> GETRANGE greeting -1 -5        # 不支持回绕操作
""

redis> GETRANGE greeting -3 -1        # 负数索引
"end"

redis> GETRANGE greeting 0 -1         # 从第一个到最后一个
"hello, my friend"

redis> GETRANGE greeting 0 1008611    # 值域范围不超过实际字符串，超过部分自动被符略
"hello, my friend"
```

# GETSET

### GETSET key value

将给定 `key` 的值设为 `value` ，并返回 `key` 的旧值(old value)。

当 `key` 存在但不是字符串类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

返回给定 `key` 的旧值。当 `key` 没有旧值时，也即是， `key` 不存在时，返回 `nil` 。

```
redis> GETSET db mongodb    # 没有旧值，返回 nil
(nil)

redis> GET db
"mongodb"

redis> GETSET db redis      # 返回旧值 mongodb
"mongodb"

redis> GET db
"redis"
```

# 模式

GETSET 可以和 *INCR* 组合使用，实现一个有原子性(atomic)复位操作的计数器(counter)。

举例来说，每次当某个事件发生时，进程可能对一个名为 `mycount` 的 `key` 调用 *INCR* 操作，通常我们还要在一个原子时间内同时完成获得计数器的值和将计数器值复位为 `0` 两个操作。

可以用命令 `GETSET mycounter 0` 来实现这一目标。

```
redis> INCR mycount
(integer) 11

redis> GETSET mycount 0  # 一个原子内完成 GET mycount 和 SET mycount 0 操作
"11"

redis> GET mycount       # 计数器被重置
"0"
```

# INCR

**INCR key**

将 `key` 中储存的数字值增一。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 INCR 操作。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

Note

这是一个针对字符串的操作，因为 Redis 没有专用的整数类型，所以 key 内储存的字符串被解释为十进制 64 位有符号整数来执行 INCR 操作。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 INCR 命令之后 `key` 的值。

```
redis> SET page_view 20
OK

redis> INCR page_view
(integer) 21

redis> GET page_view    # 数字值在 Redis 中以字符串的形式保存
"21"
```

# 模式：计数器

计数器是 Redis 的原子性自增操作可实现的最直观的模式了，它的想法相当简单：每当某个操作发生时，向 Redis 发送一个 INCR 命令。

比如在一个 web 应用程序中，如果想知道用户在一年中每天的点击量，那么只要将用户 ID 以及相关的日期信息作为键，并在每次用户点击页面时，执行一次自增操作即可。

比如用户名是 `peter` ，点击时间是 2012 年 3 月 22 日，那么执行命令 `INCR peter::2012.3.22` 。

可以用以下几种方式扩展这个简单的模式：

- 可以通过组合使用 INCR 和 *EXPIRE* ，来达到只在规定的生存时间内进行计数(counting)的目的。
- 客户端可以通过使用 *GETSET* 命令原子性地获取计数器的当前值并将计数器清零，更多信息请参考 *GETSET* 命令。
- 使用其他自增/自减操作，比如 *DECR* 和 *INCRBY* ，用户可以通过执行不同的操作增加或减少计数器的值，比如在游戏中的记分器就可能用到这些命令。

# 模式：限速器

限速器是特殊化的计算器，它用于限制一个操作可以被执行的速率(rate)。

限速器的典型用法是限制公开 API 的请求次数，以下是一个限速器实现示例，它将 API 的最大请求数限制在每个 IP 地址每秒钟十个之内：

```
FUNCTION LIMIT_API_CALL(ip)
ts = CURRENT_UNIX_TIME()
keyname = ip+":"+ts
current = GET(keyname)

IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
END

IF current == NULL THEN
    MULTI
        INCR(keyname, 1)
        EXPIRE(keyname, 1)
    EXEC
ELSE
    INCR(keyname, 1)
END

PERFORM_API_CALL()
```

这个实现每秒钟为每个 IP 地址使用一个不同的计数器，并用 *EXPIRE* 命令设置生存时间(这样 Redis 就会负责自动删除过期的计数器)。

注意，我们使用事务打包执行 *INCR* 命令和 *EXPIRE* 命令，避免引入竞争条件，保证每次调用 API 时都可以正确地对计数器进行自增操作并设置生存时间。

以下是另一个限速器实现：

```
FUNCTION LIMIT_API_CALL(ip):
current = GET(ip)
IF current != NULL AND current > 10 THEN
    ERROR "too many requests per second"
ELSE
    value = INCR(ip)
    IF value == 1 THEN
        EXPIRE(ip,1)
    END
    PERFORM_API_CALL()
END
```

这个限速器只使用单个计数器，它的生存时间为一秒钟，如果在一秒钟内，这个计数器的值大于 `10` 的话，那么访问就会被禁止。

这个新的限速器在思路方面是没有问题的，但它在实现方面不够严谨，如果我们仔细观察一下的话，就会发现在 *INCR* 和 *EXPIRE* 之间存在着一个竞争条件，假如客户端在执行 *INCR* 之后，因为某些原因(比如客户端失败)而忘记设置 *EXPIRE* 的话，那么这个计数器就会一直存在下去，造成每个用户只能访问 `10` 次，噢，这简直是个灾难！

要消灭这个实现中的竞争条件，我们可以将它转化为一个 Lua 脚本，并放到 Redis 中运行(这个方法仅限于 Redis 2.6 及以上的版本)：

```
local current
current = redis.call("incr",KEYS[1])
if tonumber(current) == 1 then
    redis.call("expire",KEYS[1],1)
end
```

通过将计数器作为脚本放到 Redis 上运行，我们保证了 *INCR* 和 *EXPIRE* 两个操作的原子性，现在这个脚本实现不会引入竞争条件，它可以运作的很好。

关于在 Redis 中运行 Lua 脚本的更多信息，请参考 *EVAL* 命令。

还有另一种消灭竞争条件的方法，就是使用 Redis 的列表结构来代替 *INCR* 命令，这个方法无须脚本支持，因此它在 Redis 2.6 以下的版本也可以运行得很好：

```
FUNCTION LIMIT_API_CALL(ip)
current = LLEN(ip)
IF current > 10 THEN
    ERROR "too many requests per second"
ELSE
    IF EXISTS(ip) == FALSE
        MULTI
            RPUSH(ip,ip)
            EXPIRE(ip,1)
        EXEC
    ELSE
        RPUSHX(ip,ip)
    END
    PERFORM_API_CALL()
END
```

新的限速器使用了列表结构作为容器， *LLEN* 用于对访问次数进行检查，一个事务包裹着 *RPUSH* 和 *EXPIRE* 两个命令，用于在第一次执行计数时创建列表，并正确设置地设置过期时间，最后， *RPUSHX* 在后续的计数操作中进行增加操作。

# INCRBY

**INCRBY key increment**

将 `key` 所储存的值加上增量 `increment` 。

如果 `key` 不存在，那么 `key` 的值会先被初始化为 `0` ，然后再执行 INCRBY 命令。

如果值包含错误的类型，或字符串类型的值不能表示为数字，那么返回一个错误。

本操作的值限制在 64 位(bit)有符号数字表示之内。

关于递增(increment) / 递减(decrement)操作的更多信息，参见 *INCR* 命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

加上 `increment` 之后， `key` 的值。

```
# key 存在且是数字值

redis> SET rank 50
OK

redis> INCRBY rank 20
(integer) 70

redis> GET rank
"70"

# key 不存在时

redis> EXISTS counter
(integer) 0

redis> INCRBY counter 30
(integer) 30

redis> GET counter
"30"

# key 不是数字值时

redis> SET book "long long ago..."
OK

redis> INCRBY book 200
(error) ERR value is not an integer or out of range
```

# INCRBYFLOAT

**INCRBYFLOAT key increment**

为 `key` 中所储存的值加上浮点数增量 `increment` 。

如果 `key` 不存在，那么 INCRBYFLOAT 会先将 `key` 的值设为 `0` ，再执行加法操作。

如果命令执行成功，那么 `key` 的值会被更新为（执行加法之后的）新值，并且新值会以字符串的形式返回给调用者。

无论是 `key` 的值，还是增量 `increment` ，都可以使用像 `2.0e7` 、 `3e5` 、 `90e-2` 那样的指数符号(exponential notation)来表示，但是，执行 **INCRBYFLOAT** 命令之后的值总是以同样的形式储存，也即是，它们总是由一个数字，一个（可选的）小数点和一个任意位的小数部分组成（比如 `3.14` 、 `69.768` ，诸如此类)，小数部分尾随的 `0` 会被移除，如果有需要的话，还会将浮点数改为整数（比如 `3.0` 会被保存成 `3` ）。

除此之外，无论加法计算所得的浮点数的实际精度有多长， INCRBYFLOAT 的计算结果也最多只能表示小数点的后十七位。

当以下任意一个条件发生时，返回一个错误：

- `key` 的值不是字符串类型(因为 Redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型）
- `key` 当前的值或者给定的增量 `increment` 不能解释(parse)为双精度浮点数(double precision floating point number）

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行命令之后 `key` 的值。

```
# 值和增量都不是指数符号

redis> SET mykey 10.50
OK

redis> INCRBYFLOAT mykey 0.1
"10.6"

# 值和增量都是指数符号

redis> SET mykey 314e-2
OK

redis> GET mykey                # 用 SET 设置的值可以是指数符号
"314e-2"

redis> INCRBYFLOAT mykey 0      # 但执行 INCRBYFLOAT 之后格式会被改成非指数符号
"3.14"

# 可以对整数类型执行

redis> SET mykey 3
OK

redis> INCRBYFLOAT mykey 1.1
"4.1"

# 后跟的 0 会被移除

redis> SET mykey 3.0
OK

redis> GET mykey                                    # SET 设置的值小数部分可以是 0
"3.0"

redis> INCRBYFLOAT mykey 1.000000000000000000000    # 但 INCRBYFLOAT 会将无用的 0 忽略掉，有
"4"

redis> GET mykey
"4"
```

# MGET

### MGET key [key ...]

返回所有(一个或多个)给定 `key` 的值。

如果给定的 `key` 里面，有某个 `key` 不存在，那么这个 `key` 返回特殊值 `nil` 。因此，该命令永不失败。

可用版本：

>= 1.0.0

时间复杂度:

O(N) , `N` 为给定 `key` 的数量。

返回值：

一个包含所有给定 `key` 的值的列表。

```
redis> SET redis redis.com
OK

redis> SET mongodb mongodb.org
OK

redis> MGET redis mongodb
1) "redis.com"
2) "mongodb.org"

redis> MGET redis mongodb mysql     # 不存在的 mysql 返回 nil
1) "redis.com"
2) "mongodb.org"
3) (nil)
```

# MSET

### MSET key value [key value ...]

同时设置一个或多个 `key-value` 对。

如果某个给定 `key` 已经存在，那么 MSET 会用新值覆盖原来的旧值，如果这不是你所希望的效果，请考虑使用 *MSETNX* 命令：它只会在所有给定 `key` 都不存在的情况下进行设置操作。

MSET 是一个原子性(atomic)操作，所有给定 `key` 都会在同一时间内被设置，某些给定 `key` 被更新而另一些给定 `key` 没有改变的情况，不可能发生。

可用版本：

>= 1.0.1

时间复杂度：

O(N)， `N` 为要设置的 `key` 数量。

返回值：

总是返回 `OK` (因为 `MSET` 不可能失败)

```
redis> MSET date "2012.3.30" time "11:00 a.m." weather "sunny"
OK

redis> MGET date time weather
1) "2012.3.30"
2) "11:00 a.m."
3) "sunny"

# MSET 覆盖旧值例子

redis> SET google "google.hk"
OK

redis> MSET google "google.com"
OK

redis> GET google
"google.com"
```

# MSETNX

**MSETNX key value [key value ...]**

同时设置一个或多个 `key-value` 对，当且仅当所有给定 `key` 都不存在。

即使只有一个给定 `key` 已存在， MSETNX 也会拒绝执行所有给定 `key` 的设置操作。

MSETNX 是原子性的，因此它可以用作设置多个不同 `key` 表示不同字段(field)的唯一性逻辑对象(unique logic object)，所有字段要么全被设置，要么全不被设置。

可用版本：

>= 1.0.1

时间复杂度：

O(N)， `N` 为要设置的 `key` 的数量。

返回值：

当所有 `key` 都成功设置，返回 `1` 。如果所有给定 `key` 都设置失败(至少有一个 `key` 已经存在)，那么返回 `0` 。

```
# 对不存在的 key 进行 MSETNX

redis> MSETNX rmdbs "MySQL" nosql "MongoDB" key-value-store "redis"
(integer) 1

redis> MGET rmdbs nosql key-value-store
1) "MySQL"
2) "MongoDB"
3) "redis"

# MSET 的给定 key 当中有已存在的 key

redis> MSETNX rmdbs "Sqlite" language "python"  # rmdbs 键已经存在，操作失败
(integer) 0

redis> EXISTS language                          # 因为 MSET 是原子性操作，language 没有被设置
(integer) 0

redis> GET rmdbs                                # rmdbs 也没有被修改
"MySQL"
```

# PSETEX

### PSETEX key milliseconds value

这个命令和 *SETEX* 命令相似，但它以毫秒为单位设置 `key` 的生存时间，而不是像 *SETEX* 命令那样，以秒为单位。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

设置成功时返回 `OK` 。

```
redis> PSETEX mykey 1000 "Hello"
OK

redis> PTTL mykey
(integer) 999

redis> GET mykey
"Hello"
```

# SET

### SET key value [EX seconds] [PX milliseconds] [NX|XX]

将字符串值 `value` 关联到 `key` 。

如果 `key` 已经持有其他值， SET 就覆写旧值，无视类型。

对于某个原本带有生存时间（TTL）的键来说， 当 *SET* 命令成功在这个键上执行时， 这个键原有的 TTL 将被清除。

可选参数

从 Redis 2.6.12 版本开始， *SET* 命令的行为可以通过一系列参数来修改：

- `EX second` ：设置键的过期时间为 `second` 秒。 `SET key value EX second` 效果等同于 `SETEX key second value` 。
- `PX millisecond` ：设置键的过期时间为 `millisecond` 毫秒。 `SET key value PX millisecond` 效果等同于 `PSETEX key millisecond value` 。
- `NX` ：只在键不存在时，才对键进行设置操作。 `SET key value NX` 效果等同于 `SETNX key value` 。
- `XX` ：只在键已经存在时，才对键进行设置操作。

Note

因为 *SET* 命令可以通过参数来实现和 *SETNX* 、 *SETEX* 和 *PSETEX* 三个命令的效果，所以将来的 Redis 版本可能会废弃并最终移除 *SETNX* 、 *SETEX* 和 *PSETEX* 这三个命令。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

在 Redis 2.6.12 版本以前， *SET* 命令总是返回 `OK` 。

从 Redis 2.6.12 版本开始， *SET* 在设置操作成功完成时，才返回 `OK` 。如果设置了 `NX` 或者 `XX` ，但因为条件没达到而造成设置操作未执行，那么命令返回空批量回复（NULL Bulk Reply）。

```
# 对不存在的键进行设置

redis 127.0.0.1:6379> SET key "value"
```

```
OK

redis 127.0.0.1:6379> GET key
"value"

# 对已存在的键进行设置

redis 127.0.0.1:6379> SET key "new-value"
OK

redis 127.0.0.1:6379> GET key
"new-value"

# 使用 EX 选项

redis 127.0.0.1:6379> SET key-with-expire-time "hello" EX 10086
OK

redis 127.0.0.1:6379> GET key-with-expire-time
"hello"

redis 127.0.0.1:6379> TTL key-with-expire-time
(integer) 10069

# 使用 PX 选项

redis 127.0.0.1:6379> SET key-with-pexpire-time "moto" PX 123321
OK

redis 127.0.0.1:6379> GET key-with-pexpire-time
"moto"

redis 127.0.0.1:6379> PTTL key-with-pexpire-time
(integer) 111939

# 使用 NX 选项

redis 127.0.0.1:6379> SET not-exists-key "value" NX
OK      # 键不存在，设置成功

redis 127.0.0.1:6379> GET not-exists-key
"value"

redis 127.0.0.1:6379> SET not-exists-key "new-value" NX
(nil)   # 键已经存在，设置失败

redis 127.0.0.1:6379> GEt not-exists-key
"value" # 维持原值不变

# 使用 XX 选项

redis 127.0.0.1:6379> EXISTS exists-key
(integer) 0

redis 127.0.0.1:6379> SET exists-key "value" XX
(nil)   # 因为键不存在，设置失败

redis 127.0.0.1:6379> SET exists-key "value"
OK      # 先给键设置一个值

redis 127.0.0.1:6379> SET exists-key "new-value" XX
OK      # 设置新值成功

redis 127.0.0.1:6379> GET exists-key
"new-value"

# NX 或 XX 可以和 EX 或者 PX 组合使用

redis 127.0.0.1:6379> SET key-with-expire-and-NX "hello" EX 10086 NX
OK

redis 127.0.0.1:6379> GET key-with-expire-and-NX
```

```
"hello"

redis 127.0.0.1:6379> TTL key-with-expire-and-NX
(integer) 10063

redis 127.0.0.1:6379> SET key-with-pexpire-and-XX "old value"
OK

redis 127.0.0.1:6379> SET key-with-pexpire-and-XX "new value" PX 123321
OK

redis 127.0.0.1:6379> GET key-with-pexpire-and-XX
"new value"

redis 127.0.0.1:6379> PTTL key-with-pexpire-and-XX
(integer) 112999

# EX 和 PX 可以同时出现，但后面给出的选项会覆盖前面给出的选项

redis 127.0.0.1:6379> SET key "value" EX 1000 PX 5000000
OK

redis 127.0.0.1:6379> TTL key
(integer) 4993  # 这是 PX 参数设置的值

redis 127.0.0.1:6379> SET another-key "value" PX 5000000 EX 1000
OK

redis 127.0.0.1:6379> TTL another-key
(integer) 997   # 这是 EX 参数设置的值
```

# 使用模式

命令 `SET resource-name anystring NX EX max-lock-time` 是一种在 Redis 中实现锁的简单方法。

客户端执行以上的命令：

- 如果服务器返回 `OK` ，那么这个客户端获得锁。
- 如果服务器返回 `NIL` ，那么客户端获取锁失败，可以在稍后再重试。

设置的过期时间到达之后，锁将自动释放。

可以通过以下修改，让这个锁实现更健壮：

- 不使用固定的字符串作为键的值，而是设置一个不可猜测（non-guessable）的长随机字符串，作为口令串（token）。
- 不使用 *DEL* 命令来释放锁，而是发送一个 Lua 脚本，这个脚本只在客户端传入的值和键的口令串相匹配时，才对键进行删除。

这两个改动可以防止持有过期锁的客户端误删现有锁的情况出现。

以下是一个简单的解锁脚本示例：

```
if redis.call("get",KEYS[1]) == ARGV[1]
then
    return redis.call("del",KEYS[1])
else
    return 0
end
```

这个脚本可以通过 `EVAL ...script... 1 resource-name token-value` 命令来调用。

# SETBIT

### SETBIT key offset value

对 `key` 所储存的字符串值，设置或清除指定偏移量上的位(bit)。

位的设置或清除取决于 `value` 参数，可以是 `0` 也可以是 `1` 。

当 `key` 不存在时，自动生成一个新的字符串值。

字符串会进行伸展(grown)以确保它可以将 `value` 保存在指定的偏移量上。当字符串值进行伸展时，空白位置以 `0` 填充。

`offset` 参数必须大于或等于 `0` ，小于 2^32 (bit 映射被限制在 512 MB 之内)。

Warning

对使用大的 `offset` 的 SETBIT 操作来说，内存分配可能造成 Redis 服务器被阻塞。具体参考 *SETRANGE* 命令，warning(警告)部分。

可用版本：

>= 2.2.0

时间复杂度:

O(1)

返回值：

指定偏移量原来储存的位。

```
redis> SETBIT bit 10086 1
(integer) 0

redis> GETBIT bit 10086
(integer) 1

redis> GETBIT bit 100   # bit 默认被初始化为 0
(integer) 0
```

# SETEX

### SETEX key seconds value

将值 `value` 关联到 `key` ，并将 `key` 的生存时间设为 `seconds` (以秒为单位)。

如果 `key` 已经存在， SETEX 命令将覆写旧值。

这个命令类似于以下两个命令：

```
SET key value
EXPIRE key seconds  # 设置生存时间
```

不同之处是， SETEX 是一个原子性(atomic)操作，关联值和设置生存时间两个动作会在同一时间内完成，该命令在 Redis 用作缓存时，非常实用。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功时返回 `OK` 。当 `seconds` 参数不合法时，返回一个错误。

```
# 在 key 不存在时进行 SETEX

redis> SETEX cache_user_id 60 10086
OK

redis> GET cache_user_id  # 值
"10086"

redis> TTL cache_user_id  # 剩余生存时间
(integer) 49

# key 已经存在时，SETEX 覆盖旧值

redis> SET cd "timeless"
OK

redis> SETEX cd 3000 "goodbye my love"
OK

redis> GET cd
"goodbye my love"

redis> TTL cd
(integer) 2997
```

# SETNX

### SETNX key value

将 `key` 的值设为 `value` ，当且仅当 `key` 不存在。

若给定的 `key` 已经存在，则 SETNX 不做任何动作。

SETNX 是『SET if Not eXists』(如果不存在，则 SET)的简写。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1` 。设置失败，返回 `0` 。

```
redis> EXISTS job                # job 不存在
(integer) 0

redis> SETNX job "programmer"    # job 设置成功
(integer) 1

redis> SETNX job "code-farmer"   # 尝试覆盖 job ，失败
(integer) 0

redis> GET job                   # 没有被覆盖
"programmer"
```

# SETRANGE

**SETRANGE key offset value**

用 `value` 参数覆写(overwrite)给定 `key` 所储存的字符串值，从偏移量 `offset` 开始。

不存在的 `key` 当作空白字符串处理。

SETRANGE 命令会确保字符串足够长以便将 `value` 设置在指定的偏移量上，如果给定 `key` 原来储存的字符串长度比偏移量小(比如字符串只有 `5` 个字符长，但你设置的 `offset` 是 `10` )，那么原字符和偏移量之间的空白将用零字节(zerobytes, `"\x00"` )来填充。

注意你能使用的最大偏移量是 2^29-1(536870911) ，因为 Redis 字符串的大小被限制在 512 兆(megabytes)以内。如果你需要使用比这更大的空间，你可以使用多个 `key` 。

Warning

当生成一个很长的字符串时，Redis 需要分配内存空间，该操作有时候可能会造成服务器阻塞(block)。在2010年的Macbook Pro上，设置偏移量为 536870911(512MB 内存分配)，耗费约 300 毫秒， 设置偏移量为 134217728(128MB 内存分配)，耗费约 80 毫秒，设置偏移量 33554432(32MB 内存分配)，耗费约 30 毫秒，设置偏移量为 8388608(8MB 内存分配)，耗费约 8 毫秒。 注意若首次内存分配成功之后，再对同一个 `key` 调用 SETRANGE 操作，无须再重新内存。

可用版本：

>= 2.2.0

时间复杂度：

对小(small)的字符串，平摊复杂度O(1)。(关于什么字符串是”小”的，请参考 *APPEND* 命令) 否则为O(M)， `M` 为 `value` 参数的长度。

返回值：

被 SETRANGE 修改之后，字符串的长度。

```
# 对非空字符串进行 SETRANGE

redis> SET greeting "hello world"
OK

redis> SETRANGE greeting 6 "Redis"
(integer) 11

redis> GET greeting
"hello Redis"

# 对空字符串/不存在的 key 进行 SETRANGE

redis> EXISTS empty_string
(integer) 0

redis> SETRANGE empty_string 5 "Redis!"   # 对不存在的 key 使用 SETRANGE
(integer) 11

redis> GET empty_string                   # 空白处被"\x00"填充
"\x00\x00\x00\x00\x00Redis!"
```

# 模式

因为有了 SETRANGE 和 *GETRANGE* 命令，你可以将 Redis 字符串用作具有O(1)随机访问时间的线性数组，这在很多真实用例中都是非常快速且高效的储存方式，具体请参考 *APPEND* 命令的『模式：时间序列』部分。

# STRLEN

**STRLEN key**

返回 `key` 所储存的字符串值的长度。

当 `key` 储存的不是字符串值时，返回一个错误。

可用版本：

>= 2.2.0

复杂度：

O(1)

返回值：

字符串值的长度。当 `key` 不存在时，返回 `0` 。

```
# 获取字符串的长度

redis> SET mykey "Hello world"
OK

redis> STRLEN mykey
(integer) 11

# 不存在的 key 长度为 0

redis> STRLEN nonexisting
(integer) 0
```

# Hash（哈希表）

# HDEL

**HDEL key field [field ...]**

删除哈希表 `key` 中的一个或多个指定域，不存在的域将被忽略。

Note

在Redis2.4以下的版本里， HDEL 每次只能删除单个域，如果你需要在一个原子时间内删除多个域，请将命令包含在 *MULTI* / *EXEC* 块内。

可用版本：

>= 2.0.0

时间复杂度:

O(N)， `N` 为要删除的域的数量。

返回值:

被成功移除的域的数量，不包括被忽略的域。

```
# 测试数据

redis> HGETALL abbr
1) "a"
2) "apple"
3) "b"
4) "banana"
5) "c"
6) "cat"
7) "d"
8) "dog"

# 删除单个域

redis> HDEL abbr a
(integer) 1

# 删除不存在的域

redis> HDEL abbr not-exists-field
(integer) 0

# 删除多个域

redis> HDEL abbr b c
(integer) 2

redis> HGETALL abbr
1) "d"
2) "dog"
```

# HEXISTS

### HEXISTS key field

查看哈希表 `key` 中，给定域 `field` 是否存在。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果哈希表含有给定域，返回 `1` 。如果哈希表不含有给定域，或 `key` 不存在，返回 `0` 。

```
redis> HEXISTS phone myphone
(integer) 0

redis> HSET phone myphone nokia-1110
(integer) 1

redis> HEXISTS phone myphone
(integer) 1
```

# HGET

### HGET key field

返回哈希表 `key` 中给定域 `field` 的值。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

给定域的值。当给定域不存在或是给定 `key` 不存在时，返回 `nil` 。

```
# 域存在

redis> HSET site redis redis.com
(integer) 1

redis> HGET site redis
"redis.com"

# 域不存在

redis> HGET site mysql
(nil)
```

# HGETALL

### HGETALL key

返回哈希表 `key` 中，所有的域和值。

在返回值里，紧跟每个域名(field name)之后是域的值(value)，所以返回值的长度是哈希表大小的两倍。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

以列表形式返回哈希表的域和域的值。若 `key` 不存在，返回空列表。

```
redis> HSET people jack "Jack Sparrow"
(integer) 1

redis> HSET people gump "Forrest Gump"
(integer) 1

redis> HGETALL people
1) "jack"          # 域
2) "Jack Sparrow"  # 值
3) "gump"
4) "Forrest Gump"
```

# HINCRBY

**HINCRBY key field increment**

为哈希表 `key` 中的域 `field` 的值加上增量 `increment` 。

增量也可以为负数，相当于对给定域进行减法操作。

如果 `key` 不存在，一个新的哈希表被创建并执行 HINCRBY 命令。

如果域 `field` 不存在，那么在执行命令前，域的值被初始化为 `0` 。

对一个储存字符串值的域 `field` 执行 HINCRBY 命令将造成一个错误。

本操作的值被限制在 64 位(bit)有符号数字表示之内。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

执行 HINCRBY 命令之后，哈希表 `key` 中域 `field` 的值。

```
# increment 为正数

redis> HEXISTS counter page_view    # 对空域进行设置
(integer) 0

redis> HINCRBY counter page_view 200
(integer) 200

redis> HGET counter page_view
"200"

# increment 为负数

redis> HGET counter page_view
"200"

redis> HINCRBY counter page_view -50
(integer) 150

redis> HGET counter page_view
"150"

# 尝试对字符串值的域执行HINCRBY命令

redis> HSET myhash string hello,world       # 设定一个字符串值
(integer) 1

redis> HGET myhash string
"hello,world"

redis> HINCRBY myhash string 1              # 命令执行失败，错误。
(error) ERR hash value is not an integer

redis> HGET myhash string                   # 原值不变
"hello,world"
```

# HINCRBYFLOAT

### HINCRBYFLOAT key field increment

为哈希表 `key` 中的域 `field` 加上浮点数增量 `increment` 。

如果哈希表中没有域 `field` ，那么 HINCRBYFLOAT 会先将域 `field` 的值设为 `0` ，然后再执行加法操作。

如果键 `key` 不存在，那么 HINCRBYFLOAT 会先创建一个哈希表，再创建域 `field` ，最后再执行加法操作。

当以下任意一个条件发生时，返回一个错误：

- 域 `field` 的值不是字符串类型(因为 redis 中的数字和浮点数都以字符串的形式保存，所以它们都属于字符串类型）
- 域 `field` 当前的值或给定的增量 `increment` 不能解释(parse)为双精度浮点数(double precision floating point number)

HINCRBYFLOAT 命令的详细功能和 *INCRBYFLOAT* 命令类似，请查看 *INCRBYFLOAT* 命令获取更多相关信息。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行加法操作之后 `field` 域的值。

```
# 值和增量都是普通小数

redis> HSET mykey field 10.50
(integer) 1
redis> HINCRBYFLOAT mykey field 0.1
"10.6"

# 值和增量都是指数符号

redis> HSET mykey field 5.0e3
(integer) 0
redis> HINCRBYFLOAT mykey field 2.0e2
"5200"

# 对不存在的键执行 HINCRBYFLOAT

redis> EXISTS price
(integer) 0
redis> HINCRBYFLOAT price milk 3.5
"3.5"
redis> HGETALL price
1) "milk"
2) "3.5"

# 对不存在的域进行 HINCRBYFLOAT

redis> HGETALL price
1) "milk"
2) "3.5"
redis> HINCRBYFLOAT price coffee 4.5   # 新增 coffee 域
"4.5"
redis> HGETALL price
1) "milk"
2) "3.5"
3) "coffee"
4) "4.5"
```

# HKEYS

**HKEYS key**

返回哈希表 `key` 中的所有域。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

一个包含哈希表中所有域的表。当 `key` 不存在时，返回一个空表。

```
# 哈希表非空

redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HKEYS website
1) "google"
2) "yahoo"

# 空哈希表/key不存在

redis> EXISTS fake_key
(integer) 0

redis> HKEYS fake_key
(empty list or set)
```

# HLEN

### HLEN key

返回哈希表 `key` 中域的数量。

时间复杂度：

O(1)

返回值：

哈希表中域的数量。当 `key` 不存在时，返回 `0` 。

```
redis> HSET db redis redis.com
(integer) 1

redis> HSET db mysql mysql.com
(integer) 1

redis> HLEN db
(integer) 2

redis> HSET db mongodb mongodb.org
(integer) 1

redis> HLEN db
(integer) 3
```

# HMGET

**HMGET key field [field ...]**

返回哈希表 `key` 中，一个或多个给定域的值。

如果给定的域不存在于哈希表，那么返回一个 `nil` 值。

因为不存在的 `key` 被当作一个空哈希表来处理，所以对一个不存在的 `key` 进行 HMGET 操作将返回一个只带有 `nil` 值的表。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为给定域的数量。

返回值：

一个包含多个给定域的关联值的表，表值的排列顺序和给定域参数的请求顺序一样。

```
redis> HMSET pet dog "doudou" cat "nounou"    # 一次设置多个域
OK

redis> HMGET pet dog cat fake_pet             # 返回值的顺序和传入参数的顺序一样
1) "doudou"
2) "nounou"
3) (nil)                                      # 不存在的域返回nil值
```

# HMSET

**HMSET key field value [field value ...]**

同时将多个 `field-value` (域-值)对设置到哈希表 `key` 中。

此命令会覆盖哈希表中已存在的域。

如果 `key` 不存在，一个空哈希表被创建并执行 HMSET 操作。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为 `field-value` 对的数量。

返回值：

如果命令执行成功，返回 `OK` 。当 `key` 不是哈希表(hash)类型时，返回一个错误。

```
redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HGET website google
"www.google.com"

redis> HGET website yahoo
"www.yahoo.com"
```

# HSET

### HSET key field value

将哈希表 `key` 中的域 `field` 的值设为 `value` 。

如果 `key` 不存在，一个新的哈希表被创建并进行 HSET 操作。

如果域 `field` 已经存在于哈希表中，旧值将被覆盖。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果 `field` 是哈希表中的一个新建域，并且值设置成功，返回 `1` 。如果哈希表中域 `field` 已经存在且旧值已被新值覆盖，返回 `0` 。

```
redis> HSET website google "www.g.cn"       # 设置一个新域
(integer) 1

redis> HSET website google "www.google.com" # 覆盖一个旧域
(integer) 0
```

# HSETNX

### HSETNX key field value

将哈希表 `key` 中的域 `field` 的值设置为 `value` ，当且仅当域 `field` 不存在。

若域 `field` 已经存在，该操作无效。

如果 `key` 不存在，一个新哈希表被创建并执行 HSETNX 命令。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

设置成功，返回 `1` 。如果给定域已经存在且没有操作被执行，返回 `0` 。

```
redis> HSETNX nosql key-value-store redis
(integer) 1

redis> HSETNX nosql key-value-store redis       # 操作无效，域 key-value-store 已存在
(integer) 0
```

# HVALS

**HVALS key**

返回哈希表 `key` 中所有域的值。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 为哈希表的大小。

返回值：

一个包含哈希表中所有值的表。当 `key` 不存在时，返回一个空表。

```
# 非空哈希表

redis> HMSET website google www.google.com yahoo www.yahoo.com
OK

redis> HVALS website
1) "www.google.com"
2) "www.yahoo.com"

# 空哈希表/不存在的key

redis> EXISTS not_exists
(integer) 0

redis> HVALS not_exists
(empty list or set)
```

# HSCAN

**HSCAN key cursor [MATCH pattern] [COUNT count]**

具体信息请参考 *SCAN* 命令。

# List（列表）

# BLPOP

**BLPOP key [key ...] timeout**

BLPOP 是列表的阻塞式(blocking)弹出原语。

它是 *LPOP* 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BLPOP 命令阻塞，直到等待超时或发现可弹出元素为止。

当给定多个 `key` 参数时，按参数 `key` 的先后顺序依次检查各个列表，弹出第一个非空列表的头元素。

非阻塞行为

当 BLPOP 被调用时，如果给定 `key` 内至少有一个非空列表，那么弹出遇到的第一个非空列表的头元素，并和被弹出元素所属的列表的名字一起，组成结果返回给调用者。

当存在多个给定 `key` 时， BLPOP 按给定 `key` 参数排列的先后顺序，依次检查各个列表。

假设现在有 `job` 、 `command` 和 `request` 三个列表，其中 `job` 不存在， `command` 和 `request` 都持有非空列表。考虑以下命令：

`BLPOP job command request 0`

BLPOP 保证返回的元素来自 `command` ，因为它是按"查找 `job` -> 查找 `command` -> 查找 `request` "这样的顺序，第一个找到的非空列表。

```
redis> DEL job command request           # 确保key都被删除
(integer) 0

redis> LPUSH command "update system..."  # 为command列表增加一个值
(integer) 1

redis> LPUSH request "visit page"        # 为request列表增加一个值
(integer) 1

redis> BLPOP job command request 0       # job 列表为空，被跳过，紧接着 command 列表的第一个元素
1) "command"                             # 弹出元素所属的列表
2) "update system..."                    # 弹出元素所属的值
```

阻塞行为

如果所有给定 `key` 都不存在或包含空列表，那么 BLPOP 命令将阻塞连接，直到等待超时，或有另一个客户端对给定 `key` 的任意一个执行 *LPUSH* 或 *RPUSH* 命令为止。

超时参数 `timeout` 接受一个以秒为单位的数字作为值。超时参数设为 `0` 表示阻塞时间可以无限期延长(block indefinitely) 。

```
redis> EXISTS job                # 确保两个 key 都不存在
(integer) 0
redis> EXISTS command
(integer) 0

redis> BLPOP job command 300     # 因为key一开始不存在，所以操作会被阻塞，直到另一客户端对 job 或者
1) "job"                         # 这里被 push 的是 job
2) "do my home work"             # 被弹出的值
(26.26s)                         # 等待的秒数

redis> BLPOP job command 5       # 等待超时的情况
(nil)
(5.66s)                          # 等待的秒数
```

相同的**key**被多个客户端同时阻塞

相同的 `key` 可以被多个客户端同时阻塞。

不同的客户端被放进一个队列中，按『先阻塞先服务』(first-BLPOP，first-served)的顺序为 `key` 执行 BLPOP 命令。

在**MULTI/EXEC**事务中的**BLPOP**

BLPOP 可以用于流水线(pipline,批量地发送多个命令并读入多个回复)，但把它用在 *MULTI* / *EXEC* 块当中没有意义。因为这要求整个服务器被阻塞以保证块执行时的原子性，该行为阻止了其他客户端执行 *LPUSH* 或 *RPUSH* 命令。

因此，一个被包裹在 *MULTI* / *EXEC* 块内的 BLPOP 命令，行为表现得就像 *LPOP* 一样，对空列表返回 `nil` ，对非空列表弹出列表元素，不进行任何阻塞操作。

```
# 对非空列表进行操作

redis> RPUSH job programming
(integer) 1

redis> MULTI
OK

redis> BLPOP job 30
QUEUED

redis> EXEC           # 不阻塞，立即返回
1) 1) "job"
   2) "programming"

# 对空列表进行操作

redis> LLEN job      # 空列表
(integer) 0

redis> MULTI
OK

redis> BLPOP job 30
QUEUED

redis> EXEC          # 不阻塞，立即返回
1) (nil)
```

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

如果列表为空，返回一个 `nil` 。否则，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 `key` ，第二个元素是被弹出元素的值。

# 模式：事件提醒

有时候，为了等待一个新元素到达数据中，需要使用轮询的方式对数据进行探查。

另一种更好的方式是，使用系统提供的阻塞原语，在新元素到达时立即进行处理，而新元素还没到达时，就一直阻塞住，避免轮询占用资源。

对于 Redis ，我们似乎需要一个阻塞版的 *SPOP* 命令，但实际上，使用 BLPOP 或者 *BRPOP* 就能很好地解决这个问题。

使用元素的客户端(消费者)可以执行类似以下的代码：

```
LOOP forever
    WHILE SPOP(key) returns elements
        ... process elements ...
    END
    BRPOP helper_key
END
```

添加元素的客户端(消费者)则执行以下代码：

```
MULTI
    SADD key element
    LPUSH helper_key x
EXEC
```

# BRPOP

**BRPOP key [key ...] timeout**

BRPOP 是列表的阻塞式(blocking)弹出原语。

它是 *RPOP* 命令的阻塞版本，当给定列表内没有任何元素可供弹出的时候，连接将被 BRPOP 命令阻塞，直到等待超时或发现可弹出元素为止。

当给定多个 `key` 参数时，按参数 `key` 的先后顺序依次检查各个列表，弹出第一个非空列表的尾部元素。

关于阻塞操作的更多信息，请查看 *BLPOP* 命令， BRPOP 除了弹出元素的位置和 *BLPOP* 不同之外，其他表现一致。

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

假如在指定时间内没有任何元素被弹出，则返回一个 `nil` 和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 `key` ，第二个元素是被弹出元素的值。

```
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> BRPOP course 30
1) "course"             # 被弹出元素所属的列表键
2) "c++101"             # 被弹出的元素
```

# BRPOPLPUSH

### BRPOPLPUSH source destination timeout

BRPOPLPUSH 是 *RPOPLPUSH* 的阻塞版本，当给定列表 `source` 不为空时，BRPOPLPUSH 的表现和 *RPOPLPUSH* 一样。

当列表 `source` 为空时， BRPOPLPUSH 命令将阻塞连接，直到等待超时，或有另一个客户端对 `source` 执行 *LPUSH* 或 *RPUSH* 命令为止。

超时参数 `timeout` 接受一个以秒为单位的数字作为值。超时参数设为 `0` 表示阻塞时间可以无限期延长(block indefinitely) 。

更多相关信息，请参考 *RPOPLPUSH* 命令。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

假如在指定时间内没有任何元素被弹出，则返回一个 `nil` 和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素的值，第二个元素是等待时长。

```
# 非空列表

redis> BRPOPLPUSH msg reciver 500
"hello moto"                        # 弹出元素的值
(3.38s)                             # 等待时长

redis> LLEN reciver
(integer) 1

redis> LRANGE reciver 0 0
1) "hello moto"

# 空列表

redis> BRPOPLPUSH msg reciver 1
(nil)
(1.34s)
```

## 模式：安全队列

参考 *RPOPLPUSH* 命令的『安全队列』模式。

# 模式：循环列表

参考 *RPOPLPUSH* 命令的『循环列表』模式。

# LINDEX

### LINDEX key index

返回列表 `key` 中，下标为 `index` 的元素。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为到达下标 `index` 过程中经过的元素数量。因此，对列表的头元素和尾元素执行 LINDEX 命令，复杂度为O(1)。

返回值:

列表中下标为 `index` 的元素。如果 `index` 参数的值不在列表的区间范围内(out of range)，返回 `nil` 。

```
redis> LPUSH mylist "World"
(integer) 1

redis> LPUSH mylist "Hello"
(integer) 2

redis> LINDEX mylist 0
"Hello"

redis> LINDEX mylist -1
"World"

redis> LINDEX mylist 3        # index不在 mylist 的区间范围内
(nil)
```

# LINSERT

### LINSERT key BEFORE|AFTER pivot value

将值 `value` 插入到列表 `key` 当中，位于值 `pivot` 之前或之后。

当 `pivot` 不存在于列表 `key` 时，不执行任何操作。

当 `key` 不存在时， `key` 被视为空列表，不执行任何操作。

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 2.2.0

时间复杂度:

O(N)， `N` 为寻找 `pivot` 过程中经过的元素数量。

返回值:

如果命令执行成功，返回插入操作完成之后，列表的长度。如果没有找到 `pivot` ，返回 `-1` 。如果 `key` 不存在或为空列表，返回 `0` 。

```
redis> RPUSH mylist "Hello"
(integer) 1

redis> RPUSH mylist "World"
(integer) 2

redis> LINSERT mylist BEFORE "World" "There"
(integer) 3

redis> LRANGE mylist 0 -1
1) "Hello"
2) "There"
3) "World"

# 对一个非空列表插入，查找一个不存在的 pivot

redis> LINSERT mylist BEFORE "go" "let's"
(integer) -1                                    # 失败

# 对一个空列表执行 LINSERT 命令

redis> EXISTS fake_list
(integer) 0

redis> LINSERT fake_list BEFORE "nono" "gogogog"
(integer) 0                                     # 失败
```

# LLEN

### LLEN key

返回列表 `key` 的长度。

如果 `key` 不存在，则 `key` 被解释为一个空列表，返回 `0` .

如果 `key` 不是列表类型，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表 `key` 的长度。

```
# 空列表

redis> LLEN job
(integer) 0

# 非空列表

redis> LPUSH job "cook food"
(integer) 1

redis> LPUSH job "have lunch"
(integer) 2

redis> LLEN job
(integer) 2
```

# LPOP

### LPOP key

移除并返回列表 `key` 的头元素。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表的头元素。当 `key` 不存在时，返回 `nil` 。

```
redis> LLEN course
(integer) 0

redis> RPUSH course algorithm001
(integer) 1

redis> RPUSH course c++101
(integer) 2

redis> LPOP course  # 移除头元素
"algorithm001"
```

# LPUSH

**LPUSH key value [value ...]**

将一个或多个值 `value` 插入到列表 `key` 的表头

如果有多个 `value` 值，那么各个 `value` 值按从左到右的顺序依次插入到表头： 比如说，对空列表 `mylist` 执行命令 `LPUSH mylist a b c` ，列表的值将是 `c b a` ，这等同于原子性地执行 `LPUSH mylist a` 、 `LPUSH mylist b` 和 `LPUSH mylist c` 三个命令。

如果 `key` 不存在，一个空列表会被创建并执行 LPUSH 操作。

当 `key` 存在但不是列表类型时，返回一个错误。

Note

在Redis 2.4版本以前的 LPUSH 命令，都只接受单个 `value` 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 LPUSH 命令后，列表的长度。

```
# 加入单个元素

redis> LPUSH languages python
(integer) 1

# 加入重复元素

redis> LPUSH languages python
(integer) 2

redis> LRANGE languages 0 -1     # 列表允许重复元素
1) "python"
2) "python"

# 加入多个元素

redis> LPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "c"
2) "b"
3) "a"
```

# LRANGE

### LRANGE key start stop

返回列表 `key` 中指定区间内的元素，区间以偏移量 `start` 和 `stop` 指定。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

注意**LRANGE**命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表，对该列表执行 `LRANGE list 0 10` ，结果是一个包含11个元素的列表，这表明 `stop` 下标也在 LRANGE 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如Ruby的 `Range.new` 、 `Array#slice` 和Python的 `range()` 函数。

超出范围的下标

超出范围的下标值不会引起错误。

如果 `start` 下标比列表的最大下标 `end` ( `LLEN list` 减去 `1` )还要大，那么 LRANGE 返回一个空列表。

如果 `stop` 下标比 `end` 下标还要大，Redis将 `stop` 的值设置为 `end` 。

可用版本：

>= 1.0.0

时间复杂度:

O(S+N)， `S` 为偏移量 `start` ， `N` 为指定区间内元素的数量。

返回值:

一个列表，包含指定区间内的元素。

```
redis> RPUSH fp-language lisp
(integer) 1

redis> LRANGE fp-language 0 0
1) "lisp"

redis> RPUSH fp-language scheme
(integer) 2

redis> LRANGE fp-language 0 1
1) "lisp"
2) "scheme"
```

# LREM

**LREM key count value**

根据参数 `count` 的值，移除列表中与参数 `value` 相等的元素。

`count` 的值可以是以下几种：

- `count > 0` : 从表头开始向表尾搜索，移除与 `value` 相等的元素，数量为 `count` 。
- `count < 0` : 从表尾开始向表头搜索，移除与 `value` 相等的元素，数量为 `count` 的绝对值。
- `count = 0` : 移除表中所有与 `value` 相等的值。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为列表的长度。

返回值：

被移除元素的数量。因为不存在的 `key` 被视作空表(empty list)，所以当 `key` 不存在时，LREM 命令总是返回 `0` 。

```
# 先创建一个表，内容排列是
# morning hello morning helllo morning

redis> LPUSH greet "morning"
(integer) 1
redis> LPUSH greet "hello"
(integer) 2
redis> LPUSH greet "morning"
(integer) 3
redis> LPUSH greet "hello"
(integer) 4
redis> LPUSH greet "morning"
(integer) 5

redis> LRANGE greet 0 4         # 查看所有元素
1) "morning"
2) "hello"
3) "morning"
4) "hello"
5) "morning"

redis> LREM greet 2 morning     # 移除从表头到表尾，最先发现的两个 morning
(integer) 2                     # 两个元素被移除

redis> LLEN greet               # 还剩 3 个元素
(integer) 3

redis> LRANGE greet 0 2
1) "hello"
2) "hello"
3) "morning"

redis> LREM greet -1 morning    # 移除从表尾到表头，第一个 morning
(integer) 1

redis> LLEN greet               # 剩下两个元素
(integer) 2

redis> LRANGE greet 0 1
1) "hello"
2) "hello"

redis> LREM greet 0 hello      # 移除表中所有 hello
(integer) 2                    # 两个 hello 被移除

redis> LLEN greet
(integer) 0
```

# LSET

### LSET key index value

将列表 `key` 下标为 `index` 的元素的值设置为 `value` 。

当 `index` 参数超出范围，或对一个空列表( `key` 不存在)进行 LSET 时，返回一个错误。

关于列表下标的更多信息，请参考 *LINDEX* 命令。

可用版本：

>= 1.0.0

时间复杂度：

对头元素或尾元素进行 LSET 操作，复杂度为 O(1)。其他情况下，为 O(N)， `N` 为列表的长度。

返回值：

操作成功返回 `ok` ，否则返回错误信息。

```
# 对空列表(key 不存在)进行 LSET

redis> EXISTS list
(integer) 0

redis> LSET list 0 item
(error) ERR no such key

# 对非空列表进行 LSET

redis> LPUSH job "cook food"
(integer) 1

redis> LRANGE job 0 0
1) "cook food"

redis> LSET job 0 "play game"
OK

redis> LRANGE job  0 0
1) "play game"

# index 超出范围

redis> LLEN list                    # 列表长度为 1
(integer) 1

redis> LSET list 3 'out of range'
(error) ERR index out of range
```

# LTRIM

### LTRIM key start stop

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

举个例子，执行命令 `LTRIM list 0 2` ，表示只保留列表 `list` 的前三个元素，其余元素全部删除。

下标(index)参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示列表的第一个元素，以 `1` 表示列表的第二个元素，以此类推。

你也可以使用负数下标，以 `-1` 表示列表的最后一个元素， `-2` 表示列表的倒数第二个元素，以此类推。

当 `key` 不是列表类型时，返回一个错误。

LTRIM 命令通常和 *LPUSH* 命令或 *RPUSH* 命令配合使用，举个例子：

```
LPUSH log newest_log
LTRIM log 0 99
```

这个例子模拟了一个日志程序，每次将最新日志 `newest_log` 放到 `log` 列表中，并且只保留最新的 `100` 项。注意当这样使用 `LTRIM` 命令时，时间复杂度是O(1)，因为平均情况下，每次只有一个元素被移除。

注意LTRIM命令和编程语言区间函数的区别

假如你有一个包含一百个元素的列表 `list` ，对该列表执行 `LTRIM list 0 10` ，结果是一个包含11个元素的列表，这表明 `stop` 下标也在 LTRIM 命令的取值范围之内(闭区间)，这和某些语言的区间函数可能不一致，比如Ruby的 `Range.new` 、 `Array#slice` 和Python的 `range()` 函数。

超出范围的下标

超出范围的下标值不会引起错误。

如果 `start` 下标比列表的最大下标 `end` ( `LLEN list` 减去 `1` )还要大，或者 `start > stop` ， LTRIM 返回一个空列表(因为 LTRIM 已经将整个列表清空)。

如果 `stop` 下标比 `end` 下标还要大，Redis将 `stop` 的值设置为 `end` 。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为被移除的元素的数量。

返回值:

命令执行成功时，返回 `ok` 。

```
# 情况 1： 常见情况， start 和 stop 都在列表的索引范围之内

redis> LRANGE alpha 0 -1       # alpha 是一个包含 5 个字符串的列表
1) "h"
2) "e"
3) "l"
4) "l"
5) "o"

redis> LTRIM alpha 1 -1        # 删除 alpha 列表索引为 0 的元素
OK

redis> LRANGE alpha 0 -1       # "h" 被删除了
1) "e"
2) "l"
3) "l"
4) "o"

# 情况 2： stop 比列表的最大下标还要大

redis> LTRIM alpha 1 10086     # 保留 alpha 列表索引 1 至索引 10086 上的元素
OK

redis> LRANGE alpha 0 -1       # 只有索引 0 上的元素 "e" 被删除了，其他元素还在
1) "l"
2) "l"
3) "o"

# 情况 3： start 和 stop 都比列表的最大下标要大，并且 start < stop

redis> LTRIM alpha 10086 123321
OK

redis> LRANGE alpha 0 -1        # 列表被清空
(empty list or set)

# 情况 4： start 和 stop 都比列表的最大下标要大，并且 start > stop

redis> RPUSH new-alpha "h" "e" "l" "l" "o"     # 重新建立一个新列表
(integer) 5

redis> LRANGE new-alpha 0 -1
1) "h"
2) "e"
3) "l"
4) "l"
5) "o"

redis> LTRIM new-alpha 123321 10086    # 执行 LTRIM
OK

redis> LRANGE new-alpha 0 -1           # 同样被清空
(empty list or set)
```

# RPOP

### RPOP key

移除并返回列表 `key` 的尾元素。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

列表的尾元素。当 `key` 不存在时，返回 `nil` 。

```
redis> RPUSH mylist "one"
(integer) 1

redis> RPUSH mylist "two"
(integer) 2

redis> RPUSH mylist "three"
(integer) 3

redis> RPOP mylist           # 返回被弹出的元素
"three"

redis> LRANGE mylist 0 -1    # 列表剩下的元素
1) "one"
2) "two"
```

# RPOPLPUSH

### RPOPLPUSH source destination

命令 RPOPLPUSH 在一个原子时间内，执行以下两个动作：

- 将列表 `source` 中的最后一个元素(尾元素)弹出，并返回给客户端。
- 将 `source` 弹出的元素插入到列表 `destination` ，作为 `destination` 列表的的头元素。

举个例子，你有两个列表 `source` 和 `destination` ， `source` 列表有元素 `a, b, c` ， `destination` 列表有元素 `x, y, z` ，执行 `RPOPLPUSH source destination` 之后， `source` 列表包含元素 `a, b` ， `destination` 列表包含元素 `c, x, y, z` ，并且元素 `c` 会被返回给客户端。

如果 `source` 不存在，值 `nil` 被返回，并且不执行其他动作。

如果 `source` 和 `destination` 相同，则列表中的表尾元素被移动到表头，并返回该元素，可以把这种特殊情况视作列表的旋转(rotation)操作。

可用版本：

>= 1.2.0

时间复杂度：

O(1)

返回值：

被弹出的元素。

```
# source 和 destination 不同

redis> LRANGE alpha 0 -1         # 查看所有元素
1) "a"
2) "b"
3) "c"
4) "d"

redis> RPOPLPUSH alpha reciver   # 执行一次 RPOPLPUSH 看看
"d"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"
3) "c"

redis> LRANGE reciver 0 -1
1) "d"

redis> RPOPLPUSH alpha reciver   # 再执行一次，证实 RPOP 和 LPUSH 的位置正确
"c"

redis> LRANGE alpha 0 -1
1) "a"
2) "b"

redis> LRANGE reciver 0 -1
1) "c"
2) "d"

# source 和 destination 相同

redis> LRANGE number 0 -1
1) "1"
2) "2"
3) "3"
4) "4"

redis> RPOPLPUSH number number
"4"

redis> LRANGE number 0 -1           # 4 被旋转到了表头
1) "4"
2) "1"
3) "2"
4) "3"

redis> RPOPLPUSH number number
"3"

redis> LRANGE number 0 -1           # 这次是 3 被旋转到了表头
1) "3"
2) "4"
3) "1"
4) "2"
```

# 模式： 安全的队列

Redis的列表经常被用作队列(queue)，用于在不同程序之间有序地交换消息(message)。一个客户端通过 *LPUSH* 命令将消息放入队列中，而另一个客户端通过 *RPOP* 或者 *BRPOP* 命令取出队列中等待时间最长的消息。

不幸的是，上面的队列方法是『不安全』的，因为在这个过程中，一个客户端可能在取出一个消息之后崩溃，而未处理完的消息也就因此丢失。

使用 RPOPLPUSH 命令(或者它的阻塞版本 *BRPOPLPUSH* )可以解决这个问题：因为它不仅返回一个消息，同时还将这个消息添加到另一个备份列表当中，如果一切正常的话，当一个客户端完成某个消息的处理之后，可以用 *LREM* 命令将这个消息从备份表删除。

最后，还可以添加一个客户端专门用于监视备份表，它自动地将超过一定处理时限的消息重新放入队列中去(负责处理该消息的客户端可能已经崩溃)，这样就不会丢失任何消息了。

# 模式：循环列表

通过使用相同的 `key` 作为 RPOPLPUSH 命令的两个参数，客户端可以用一个接一个地获取列表元素的方式，取得列表的所有元素，而不必像 *LRANGE* 命令那样一下子将所有列表元素都从服务器传送到客户端中(两种方式的总复杂度都是 O(N))。

以上的模式甚至在以下的两个情况下也能正常工作：

- 有多个客户端同时对同一个列表进行旋转(rotating)，它们获取不同的元素，直到所有元素都被读取完，之后又从头开始。
- 有客户端在向列表尾部(右边)添加新元素。

这个模式使得我们可以很容易实现这样一类系统：有 N 个客户端，需要连续不断地对一些元素进行处理，而且处理的过程必须尽可能地快。一个典型的例子就是服务器的监控程序：它们需要在尽可能短的时间内，并行地检查一组网站，确保它们的可访问性。

注意，使用这个模式的客户端是易于扩展(scala)且安全(reliable)的，因为就算接收到元素的客户端失败，元素还是保存在列表里面，不会丢失，等到下个迭代来临的时候，别的客户端又可以继续处理这些元素了。

# RPUSH

### RPUSH key value [value ...]

将一个或多个值 `value` 插入到列表 `key` 的表尾(最右边)。

如果有多个 `value` 值，那么各个 `value` 值按从左到右的顺序依次插入到表尾：比如对一个空列表 `mylist` 执行 `RPUSH mylist a b c` ，得出的结果列表为 `a b c` ，等同于执行命令 `RPUSH mylist a` 、 `RPUSH mylist b` 、 `RPUSH mylist c` 。

如果 `key` 不存在，一个空列表会被创建并执行 RPUSH 操作。

当 `key` 存在但不是列表类型时，返回一个错误。

Note

在 Redis 2.4 版本以前的 RPUSH 命令，都只接受单个 `value` 值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

执行 RPUSH 操作后，表的长度。

```
# 添加单个元素

redis> RPUSH languages c
(integer) 1

# 添加重复元素

redis> RPUSH languages c
(integer) 2

redis> LRANGE languages 0 -1 # 列表允许重复元素
1) "c"
2) "c"

# 添加多个元素

redis> RPUSH mylist a b c
(integer) 3

redis> LRANGE mylist 0 -1
1) "a"
2) "b"
3) "c"
```

# RPUSHX

### RPUSHX key value

将值 `value` 插入到列表 `key` 的表尾，当且仅当 `key` 存在并且是一个列表。

和 *RPUSH* 命令相反，当 `key` 不存在时， RPUSHX 命令什么也不做。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

RPUSHX 命令执行之后，表的长度。

```
# key不存在

redis> LLEN greet
(integer) 0

redis> RPUSHX greet "hello"     # 对不存在的 key 进行 RPUSHX，PUSH 失败。
(integer) 0

# key 存在且是一个非空列表

redis> RPUSH greet "hi"         # 先用 RPUSH 插入一个元素
(integer) 1

redis> RPUSHX greet "hello"     # greet 现在是一个列表类型，RPUSHX 操作成功。
(integer) 2

redis> LRANGE greet 0 -1
1) "hi"
2) "hello"
```

# Set（集合）

# SADD

### SADD key member [member ...]

将一个或多个 `member` 元素加入到集合 `key` 当中，已经存在于集合的 `member` 元素将被忽略。

假如 `key` 不存在，则创建一个只包含 `member` 元素作成员的集合。

当 `key` 不是集合类型时，返回一个错误。

Note

在Redis2.4版本以前， SADD 只接受单个 `member` 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是被添加的元素的数量。

返回值:

被添加到集合中的新元素的数量，不包括被忽略的元素。

```
# 添加单个元素

redis> SADD bbs "discuz.net"
(integer) 1

# 添加重复元素

redis> SADD bbs "discuz.net"
(integer) 0

# 添加多个元素

redis> SADD bbs "tianya.cn" "groups.google.com"
(integer) 2

redis> SMEMBERS bbs
1) "discuz.net"
2) "groups.google.com"
3) "tianya.cn"
```

# SCARD

### SCARD key

返回集合 `key` 的基数(集合中元素的数量)。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值：

集合的基数。当 `key` 不存在时，返回 `0` 。

```
redis> SADD tool pc printer phone
(integer) 3

redis> SCARD tool   # 非空集合
(integer) 3

redis> DEL tool
(integer) 1

redis> SCARD tool   # 空集合
(integer) 0
```

# SDIFF

### SDIFF key [key ...]

返回一个集合的全部成员，该集合是所有给定集合之间的差集。

不存在的 `key` 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

一个包含差集成员的列表。

```
redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SDIFF peter's_movies joe's_movies
1) "bet man"
2) "start war"
```

# SDIFFSTORE

### SDIFFSTORE destination key [key ...]

这个命令的作用和 *SDIFF* 类似，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 集合已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SMEMBERS peter's_movies
1) "bet man"
2) "start war"
3) "2012"

redis> SDIFFSTORE joe_diff_peter joe's_movies peter's_movies
(integer) 2

redis> SMEMBERS joe_diff_peter
1) "hi, lady"
2) "Fast Five"
```

# SINTER

**SINTER key [key ...]**

返回一个集合的全部成员，该集合是所有给定集合的交集。

不存在的 `key` 被视为空集。

当给定集合当中有一个空集时，结果也为空集(根据集合运算定律)。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

交集成员的列表。

```
redis> SMEMBERS group_1
1) "LI LEI"
2) "TOM"
3) "JACK"

redis> SMEMBERS group_2
1) "HAN MEIMEI"
2) "JACK"

redis> SINTER group_1 group_2
1) "JACK"
```

# SINTER

**SINTER key [key ...]**

返回一个集合的全部成员，该集合是所有给定集合的交集。

不存在的 `key` 被视为空集。

当给定集合当中有一个空集时，结果也为空集(根据集合运算定律)。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

交集成员的列表。

```
redis> SMEMBERS group_1
1) "LI LEI"
2) "TOM"
3) "JACK"

redis> SMEMBERS group_2
1) "HAN MEIMEI"
2) "JACK"

redis> SINTER group_1 group_2
1) "JACK"
```

# SINTERSTORE

**SINTERSTORE destination key [key ...]**

这个命令类似于 *SINTER* 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 集合已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N * M)， `N` 为给定集合当中基数最小的集合， `M` 为给定集合的个数。

返回值:

结果集中的成员数量。

```
redis> SMEMBERS songs
1) "good bye joe"
2) "hello,peter"

redis> SMEMBERS my_songs
1) "good bye joe"
2) "falling"

redis> SINTERSTORE song_interset songs my_songs
(integer) 1

redis> SMEMBERS song_interset
1) "good bye joe"
```

# SISMEMBER

### SISMEMBER key member

判断 `member` 元素是否集合 `key` 的成员。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 `member` 元素是集合的成员，返回 `1` 。如果 `member` 元素不是集合的成员，或 `key` 不存在，返回 `0` 。

```
redis> SMEMBERS joe's_movies
1) "hi, lady"
2) "Fast Five"
3) "2012"

redis> SISMEMBER joe's_movies "bet man"
(integer) 0

redis> SISMEMBER joe's_movies "Fast Five"
(integer) 1
```

# SMEMBERS

**SMEMBERS key**

返回集合 `key` 中的所有成员。

不存在的 `key` 被视为空集合。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为集合的基数。

返回值:

集合中的所有成员。

```
# key 不存在或集合为空

redis> EXISTS not_exists_key
(integer) 0

redis> SMEMBERS not_exists_key
(empty list or set)

# 非空集合

redis> SADD language Ruby Python Clojure
(integer) 3

redis> SMEMBERS language
1) "Python"
2) "Ruby"
3) "Clojure"
```

# SMOVE

**SMOVE source destination member**

将 `member` 元素从 `source` 集合移动到 `destination` 集合。

SMOVE 是原子性操作。

如果 `source` 集合不存在或不包含指定的 `member` 元素，则 SMOVE 命令不执行任何操作，仅返回 `0` 。否则， `member` 元素从 `source` 集合中被移除，并添加到 `destination` 集合中去。

当 `destination` 集合已经包含 `member` 元素时， SMOVE 命令只是简单地将 `source` 集合中的 `member` 元素删除。

当 `source` 或 `destination` 不是集合类型时，返回一个错误。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

如果 `member` 元素被成功移除，返回 `1` 。如果 `member` 元素不是 `source` 集合的成员，并且没有任何操作对 `destination` 集合执行，那么返回 `0` 。

```
redis> SMEMBERS songs
1) "Billie Jean"
2) "Believe Me"

redis> SMEMBERS my_songs
(empty list or set)

redis> SMOVE songs my_songs "Believe Me"
(integer) 1

redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"
```

# SPOP

**SPOP key**

移除并返回集合中的一个随机元素。

如果只想获取一个随机元素，但不想该元素从集合中被移除的话，可以使用 *SRANDMEMBER* 命令。

可用版本：

>= 1.0.0

时间复杂度:

O(1)

返回值:

被移除的随机元素。当 `key` 不存在或 `key` 是空集时，返回 `nil` 。

```
redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"
3) "Redis"

redis> SPOP db
"Redis"

redis> SMEMBERS db
1) "MySQL"
2) "MongoDB"

redis> SPOP db
"MySQL"

redis> SMEMBERS db
1) "MongoDB"
```

# SRANDMEMBER

**SRANDMEMBER key [count]**

如果命令执行时，只提供了 `key` 参数，那么返回集合中的一个随机元素。

从 Redis 2.6 版本开始， SRANDMEMBER 命令接受可选的 `count` 参数：

- 如果 `count` 为正数，且小于集合基数，那么命令返回一个包含 `count` 个元素的数组，数组中的元素各不相同。如果 `count` 大于等于集合基数，那么返回整个集合。
- 如果 `count` 为负数，那么命令返回一个数组，数组中的元素可能会重复出现多次，而数组的长度为 `count` 的绝对值。

该操作和 *SPOP* 相似，但 *SPOP* 将随机元素从集合中移除并返回，而 SRANDMEMBER 则仅仅返回随机元素，而不对集合进行任何改动。

可用版本：

>= 1.0.0

时间复杂度:

只提供 `key` 参数时为 O(1) 。如果提供了 `count` 参数，那么为 O(N) ，N 为返回数组的元素个数。

返回值:

只提供 `key` 参数时，返回一个元素；如果集合为空，返回 `nil` 。如果提供了 `count` 参数，那么返回一个数组；如果集合为空，返回空数组。

```
# 添加元素

redis> SADD fruit apple banana cherry
(integer) 3

# 只给定 key 参数，返回一个随机元素

redis> SRANDMEMBER fruit
"cherry"

redis> SRANDMEMBER fruit
"apple"

# 给定 3 为 count 参数，返回 3 个随机元素
# 每个随机元素都不相同

redis> SRANDMEMBER fruit 3
1) "apple"
2) "banana"
3) "cherry"

# 给定 -3 为 count 参数，返回 3 个随机元素
# 元素可能会重复出现多次

redis> SRANDMEMBER fruit -3
1) "banana"
2) "cherry"
3) "apple"

redis> SRANDMEMBER fruit -3
1) "apple"
2) "apple"
3) "cherry"

# 如果 count 是整数，且大于等于集合基数，那么返回整个集合

redis> SRANDMEMBER fruit 10
1) "apple"
2) "banana"
3) "cherry"

# 如果 count 是负数，且 count 的绝对值大于集合的基数
# 那么返回的数组的长度为 count 的绝对值

redis> SRANDMEMBER fruit -10
1) "banana"
2) "apple"
3) "banana"
4) "cherry"
5) "apple"
6) "apple"
7) "cherry"
8) "apple"
9) "apple"
10) "banana"

# SRANDMEMBER 并不会修改集合内容

redis> SMEMBERS fruit
1) "apple"
2) "cherry"
3) "banana"

# 集合为空时返回 nil 或者空数组

redis> SRANDMEMBER not-exists
(nil)

redis> SRANDMEMBER not-eixsts 10
(empty list or set)
```

# SREM

**SREM key member [member ...]**

移除集合 `key` 中的一个或多个 `member` 元素，不存在的 `member` 元素会被忽略。

当 `key` 不是集合类型，返回一个错误。

Note

在 Redis 2.4 版本以前， SREM 只接受单个 `member` 值。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 为给定 `member` 元素的数量。

返回值:

被成功移除的元素的数量，不包括被忽略的元素。

```
# 测试数据

redis> SMEMBERS languages
1) "c"
2) "lisp"
3) "python"
4) "ruby"

# 移除单个元素

redis> SREM languages ruby
(integer) 1

# 移除不存在元素

redis> SREM languages non-exists-language
(integer) 0

# 移除多个元素

redis> SREM languages lisp python c
(integer) 3

redis> SMEMBERS languages
(empty list or set)
```

# SUNION

**SUNION key [key ...]**

返回一个集合的全部成员，该集合是所有给定集合的并集。

不存在的 `key` 被视为空集。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

并集成员的列表。

```
redis> SMEMBERS songs
1) "Billie Jean"

redis> SMEMBERS my_songs
1) "Believe Me"

redis> SUNION songs my_songs
1) "Billie Jean"
2) "Believe Me"
```

# SUNIONSTORE

### SUNIONSTORE destination key [key ...]

这个命令类似于 *SUNION* 命令，但它将结果保存到 `destination` 集合，而不是简单地返回结果集。

如果 `destination` 已经存在，则将其覆盖。

`destination` 可以是 `key` 本身。

可用版本：

>= 1.0.0

时间复杂度:

O(N)， `N` 是所有给定集合的成员数量之和。

返回值:

结果集中的元素数量。

```
redis> SMEMBERS NoSQL
1) "MongoDB"
2) "Redis"

redis> SMEMBERS SQL
1) "sqlite"
2) "MySQL"

redis> SUNIONSTORE db NoSQL SQL
(integer) 4

redis> SMEMBERS db
1) "MySQL"
2) "sqlite"
3) "MongoDB"
4) "Redis"
```

# SSCAN

**SSCAN key cursor [MATCH pattern] [COUNT count]**

详细信息请参考 *SCAN* 命令。

# SortedSet（有序集合）

# ZADD

### ZADD key score member [[score member] [score member] ...]

将一个或多个 `member` 元素及其 `score` 值加入到有序集 `key` 当中。

如果某个 `member` 已经是有序集的成员，那么更新这个 `member` 的 `score` 值，并通过重新插入这个 `member` 元素，来保证该 `member` 在正确的位置上。

`score` 值可以是整数值或双精度浮点数。

如果 `key` 不存在，则创建一个空的有序集并执行 ZADD 操作。

当 `key` 存在但不是有序集类型时，返回一个错误。

对有序集的更多介绍请参见 sorted set 。

Note

在 Redis 2.4 版本以前， ZADD 每次只能添加一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， `N` 是有序集的基数， `M` 为成功添加的新成员的数量。

返回值:

被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员。

```
# 添加单个元素

redis> ZADD page_rank 10 google.com
(integer) 1

# 添加多个元素

redis> ZADD page_rank 9 baidu.com 8 bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 添加已存在元素，且 score 值不变

redis> ZADD page_rank 10 google.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES  # 没有改变
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 添加已存在元素，但是改变 score 值

redis> ZADD page_rank 6 bing.com
(integer) 0

redis> ZRANGE page_rank 0 -1 WITHSCORES  # bing.com 元素的 score 值被改变
1) "bing.com"
2) "6"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"
```

# ZCARD

### ZCARD key

返回有序集 `key` 的基数。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

当 `key` 存在且是有序集类型时，返回有序集的基数。当 `key` 不存在时，返回 `0` 。

```
redis > ZADD salary 2000 tom    # 添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 1

redis > ZADD salary 5000 jack   # 再添加一个成员
(integer) 1

redis > ZCARD salary
(integer) 2

redis > EXISTS non_exists_key   # 对不存在的 key 进行 ZCARD 操作
(integer) 0

redis > ZCARD non_exists_key
(integer) 0
```

# ZCOUNT

### ZCOUNT key min max

返回有序集 `key` 中， `score` 值在 `min` 和 `max` 之间(默认包括 `score` 值等于 `min` 或 `max` )的成员的数量。

关于参数 `min` 和 `max` 的详细使用方法，请参考 *ZRANGEBYSCORE* 命令。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))， `N` 为有序集的基数。

返回值:

`score` 值在 `min` 和 `max` 之间的成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES    # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZCOUNT salary 2000 5000          # 计算薪水在 2000-5000 之间的人数
(integer) 3

redis> ZCOUNT salary 3000 5000          # 计算薪水在 3000-5000 之间的人数
(integer) 2
```

# ZINCRBY

### ZINCRBY key increment member

为有序集 `key` 的成员 `member` 的 `score` 值加上增量 `increment` 。

可以通过传递一个负数值 `increment` ，让 `score` 减去相应的值，比如 `ZINCRBY key -5 member` ，就是让 `member` 的 `score` 值减去 `5` 。

当 `key` 不存在，或 `member` 不是 `key` 的成员时， `ZINCRBY key increment member` 等同于 `ZADD key increment member` 。

当 `key` 不是有序集类型时，返回一个错误。

`score` 值可以是整数值或双精度浮点数。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N))

返回值:

`member` 成员的新 `score` 值，以字符串形式表示。

```
redis> ZSCORE salary tom
"2000"

redis> ZINCRBY salary 2000 tom   # tom 加薪啦！
"4000"
```

# ZRANGE

### ZRANGE key start stop [WITHSCORES]

返回有序集 `key` 中，指定区间内的成员。

其中成员的位置按 `score` 值递增(从小到大)来排序。

具有相同 `score` 值的成员按字典序(lexicographical order )来排列。

如果你需要成员按 `score` 值递减(从大到小)来排列，请使用 *ZREVRANGE* 命令。

下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。你也可以使用负数下标，以 `-1` 表示最后一个成员， `-2` 表示倒数第二个成员，以此类推。超出范围的下标并不会引起错误。比如说，当 `start` 的值比有序集的最大下标还要大，或是 `start > stop` 时， ZRANGE 命令只是简单地返回一个空列表。另一方面，假如 `stop` 参数的值比有序集的最大下标还要大，那么 Redis 将 `stop` 当作最大下标来处理。可以通过使用 `WITHSCORES` 选项，来让成员和它的 `score` 值一并返回，返回列表以 `value1,score1, ..., valueN,scoreN` 的格式表示。客户端库可能会返回一些更复杂的数据类型，比如数组、元组等。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis > ZRANGE salary 0 -1 WITHSCORES             # 显示整个有序集成员
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 1 2 WITHSCORES              # 显示有序集下标区间 1 至 2 的成员
1) "tom"
2) "5000"
3) "boss"
4) "10086"

redis > ZRANGE salary 0 200000 WITHSCORES         # 测试 end 下标超出最大下标时的情况
1) "jack"
2) "3500"
3) "tom"
4) "5000"
5) "boss"
6) "10086"

redis > ZRANGE salary 200000 3000000 WITHSCORES   # 测试当给定区间不存在于有序集时的情况
(empty list or set)
```

# ZRANGEBYSCORE

**ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]**

返回有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max` )的成员。有序集成员按 `score` 值递增(从小到大)次序排列。

具有相同 `score` 值的成员按字典序(lexicographical order)来排列(该属性是有序集提供的，不需要额外的计算)。

可选的 `LIMIT` 参数指定返回结果的数量及区间(就像SQL中的 `SELECT LIMIT offset, count` )，注意当 `offset` 很大时，定位 `offset` 的操作可能需要遍历整个有序集，此过程最坏复杂度为 O(N) 时间。

可选的 `WITHSCORES` 参数决定结果集是单单返回有序集的成员，还是将有序集成员及其 `score` 值一起返回。该选项自 Redis 2.0 版本起可用。

区间及无限

`min` 和 `max` 可以是 `-inf` 和 `+inf` ，这样一来，你就可以在不知道有序集的最低和最高 `score` 值的情况下，使用 ZRANGEBYSCORE 这类命令。

默认情况下，区间的取值使用闭区间 (小于等于或大于等于)，你也可以通过给参数前增加 `(` 符号来使用可选的开区间 (小于或大于)。

举个例子：

```
ZRANGEBYSCORE zset (1 5
```

返回所有符合条件 `1 < score <= 5` 的成员，而

```
ZRANGEBYSCORE zset (5 (10
```

则返回所有符合条件 `5 < score < 10` 的成员。

可用版本：

>= 1.0.5

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数， `M` 为被结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis> ZADD salary 2500 jack                        # 测试数据
(integer) 0
redis> ZADD salary 5000 tom
(integer) 0
redis> ZADD salary 12000 peter
(integer) 0

redis> ZRANGEBYSCORE salary -inf +inf               # 显示整个有序集
1) "jack"
2) "tom"
3) "peter"

redis> ZRANGEBYSCORE salary -inf +inf WITHSCORES    # 显示整个有序集及成员的 score 值
1) "jack"
2) "2500"
3) "tom"
4) "5000"
5) "peter"
6) "12000"

redis> ZRANGEBYSCORE salary -inf 5000 WITHSCORES    # 显示工资 <=5000 的所有成员
1) "jack"
2) "2500"
3) "tom"
4) "5000"

redis> ZRANGEBYSCORE salary (5000 400000            # 显示工资大于 5000 小于等于 400000 的成
1) "peter"
```

# ZRANK

### ZRANK key member

返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递增(从小到大)顺序排列。

排名以 `0` 为底，也就是说， `score` 值最小的成员排名为 `0` 。

使用 *ZREVRANK* 命令可以获得成员按 `score` 值递减(从大到小)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 `member` 是有序集 `key` 的成员，返回 `member` 的排名。如果 `member` 不是有序集 `key` 的成员，返回 `nil` 。

```
redis> ZRANGE salary 0 -1 WITHSCORES        # 显示所有成员及其 score 值
1) "peter"
2) "3500"
3) "tom"
4) "4000"
5) "jack"
6) "5000"

redis> ZRANK salary tom                     # 显示 tom 的薪水排名，第二
(integer) 1
```

# ZREM

**ZREM key member [member ...]**

移除有序集 `key` 中的一个或多个成员，不存在的成员将被忽略。

当 `key` 存在但不是有序集类型时，返回一个错误。

Note

在 Redis 2.4 版本以前， ZREM 每次只能删除一个元素。

可用版本：

>= 1.2.0

时间复杂度:

O(M*log(N))， `N` 为有序集的基数， `M` 为被成功移除的成员的数量。

返回值:

被成功移除的成员的数量，不包括被忽略的成员。

```
# 测试数据

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"
5) "google.com"
6) "10"

# 移除单个元素

redis> ZREM page_rank google.com
(integer) 1

redis> ZRANGE page_rank 0 -1 WITHSCORES
1) "bing.com"
2) "8"
3) "baidu.com"
4) "9"

# 移除多个元素

redis> ZREM page_rank baidu.com bing.com
(integer) 2

redis> ZRANGE page_rank 0 -1 WITHSCORES
(empty list or set)

# 移除不存在元素

redis> ZREM page_rank non-exists-element
(integer) 0
```

# ZREMRANGEBYRANK

### ZREMRANGEBYRANK key start stop

移除有序集 `key` 中，指定排名(rank)区间内的所有成员。

区间分别以下标参数 `start` 和 `stop` 指出，包含 `start` 和 `stop` 在内。

下标参数 `start` 和 `stop` 都以 `0` 为底，也就是说，以 `0` 表示有序集第一个成员，以 `1` 表示有序集第二个成员，以此类推。你也可以使用负数下标，以 `-1` 表示最后一个成员，`-2` 表示倒数第二个成员，以此类推。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZADD salary 2000 jack
(integer) 1
redis> ZADD salary 5000 tom
(integer) 1
redis> ZADD salary 3500 peter
(integer) 1

redis> ZREMRANGEBYRANK salary 0 1       # 移除下标 0 至 1 区间内的成员
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES    # 有序集只剩下一个成员
1) "tom"
2) "5000"
```

# ZREMRANGEBYSCORE

**ZREMRANGEBYSCORE key min max**

移除有序集 `key` 中，所有 `score` 值介于 `min` 和 `max` 之间(包括等于 `min` 或 `max` )的成员。

自版本2.1.6开始， `score` 值等于 `min` 或 `max` 的成员也可以不包括在内，详情请参见 *ZRANGEBYSCORE* 命令。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为被移除成员的数量。

返回值:

被移除成员的数量。

```
redis> ZRANGE salary 0 -1 WITHSCORES          # 显示有序集内所有成员及其 score 值
1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"

redis> ZREMRANGEBYSCORE salary 1500 3500      # 移除所有薪水在 1500 到 3500 内的员工
(integer) 2

redis> ZRANGE salary 0 -1 WITHSCORES          # 剩下的有序集成员
1) "jack"
2) "5000"
```

# ZREVRANGE

### ZREVRANGE key start stop [WITHSCORES]

返回有序集 `key` 中，指定区间内的成员。

其中成员的位置按 `score` 值递减(从大到小)来排列。具有相同 `score` 值的成员按字典序的逆序(reverse lexicographical order)排列。

除了成员按 `score` 值递减的次序排列这一点外， ZREVRANGE 命令的其他方面和 *ZRANGE* 命令一样。

可用版本：

>= 1.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数，而 `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis> ZRANGE salary 0 -1 WITHSCORES        # 递增排列
1) "peter"
2) "3500"
3) "tom"
4) "4000"
5) "jack"
6) "5000"

redis> ZREVRANGE salary 0 -1 WITHSCORES     # 递减排列
1) "jack"
2) "5000"
3) "tom"
4) "4000"
5) "peter"
6) "3500"
```

# ZREVRANGEBYSCORE

### ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]

返回有序集 `key` 中， `score` 值介于 `max` 和 `min` 之间(默认包括等于 `max` 或 `min` )的所有的成员。有序集成员按 `score` 值递减(从大到小)的次序排列。

具有相同 `score` 值的成员按字典序的逆序(reverse lexicographical order )排列。

除了成员按 `score` 值递减的次序排列这一点外， ZREVRANGEBYSCORE 命令的其他方面和 *ZRANGEBYSCORE* 命令一样。

可用版本：

>= 2.2.0

时间复杂度:

O(log(N)+M)， `N` 为有序集的基数， `M` 为结果集的基数。

返回值:

指定区间内，带有 `score` 值(可选)的有序集成员的列表。

```
redis > ZADD salary 10086 jack
(integer) 1
redis > ZADD salary 5000 tom
(integer) 1
redis > ZADD salary 7500 peter
(integer) 1
redis > ZADD salary 3500 joe
(integer) 1

redis > ZREVRANGEBYSCORE salary +inf -inf   # 逆序排列所有成员
1) "jack"
2) "peter"
3) "tom"
4) "joe"

redis > ZREVRANGEBYSCORE salary 10000 2000  # 逆序排列薪水介于 10000 和 2000 之间的成员
1) "peter"
2) "tom"
3) "joe"
```

# ZREVRANK

### ZREVRANK key member

返回有序集 `key` 中成员 `member` 的排名。其中有序集成员按 `score` 值递减(从大到小)排序。

排名以 `0` 为底，也就是说， `score` 值最大的成员排名为 `0` 。

使用 *ZRANK* 命令可以获得成员按 `score` 值递增(从小到大)排列的排名。

可用版本：

>= 2.0.0

时间复杂度:

O(log(N))

返回值:

如果 `member` 是有序集 `key` 的成员，返回 `member` 的排名。如果 `member` 不是有序集 `key` 的成员，返回 `nil` 。

```
redis 127.0.0.1:6379> ZRANGE salary 0 -1 WITHSCORES     # 测试数据
1) "jack"
2) "2000"
3) "peter"
4) "3500"
5) "tom"
6) "5000"

redis> ZREVRANK salary peter     # peter 的工资排第二
(integer) 1

redis> ZREVRANK salary tom       # tom 的工资最高
(integer) 0
```

# ZSCORE

### ZSCORE key member

返回有序集 `key` 中，成员 `member` 的 `score` 值。

如果 `member` 元素不是有序集 `key` 的成员，或 `key` 不存在，返回 `nil` 。

可用版本：

>= 1.2.0

时间复杂度:

O(1)

返回值:

`member` 成员的 `score` 值，以字符串形式表示。

```
redis> ZRANGE salary 0 -1 WITHSCORES    # 测试数据
1) "tom"
2) "2000"
3) "peter"
4) "3500"
5) "jack"
6) "5000"

redis> ZSCORE salary peter              # 注意返回值是字符串
"3500"
```

# ZUNIONSTORE

### ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

计算给定的一个或多个有序集的并集，其中给定 `key` 的数量必须以 `numkeys` 参数指定，并将该并集(结果集)储存到 `destination` 。

默认情况下，结果集中某个成员的 `score` 值是所有给定集下该成员 `score` 值之 和 。

### WEIGHTS

使用 `WEIGHTS` 选项，你可以为 每个 给定有序集 分别 指定一个乘法因子(multiplication factor)，每个给定有序集的所有成员的 `score` 值在传递给聚合函数(aggregation function)之前都要先乘以该有序集的因子。

如果没有指定 `WEIGHTS` 选项，乘法因子默认设置为 `1` 。

### AGGREGATE

使用 `AGGREGATE` 选项，你可以指定并集的结果集的聚合方式。

默认使用的参数 `SUM` ，可以将所有集合中某个成员的 `score` 值之 和 作为结果集中该成员的 `score` 值；使用参数 `MIN` ，可以将所有集合中某个成员的 最小 `score` 值作为结果集中该成员的 `score` 值；而参数 `MAX` 则是将所有集合中某个成员的 最大 `score` 值作为结果集中该成员的 `score` 值。

可用版本：

>= 2.0.0

时间复杂度:

O(N)+O(M log(M))， `N` 为给定有序集基数的总和， `M` 为结果集的基数。

返回值:

保存到 `destination` 的结果集的基数。

```
redis> ZRANGE programmer 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"

redis> ZRANGE manager 0 -1 WITHSCORES
1) "herry"
2) "2000"
3) "mary"
4) "3500"
5) "bob"
6) "4000"

redis> ZUNIONSTORE salary 2 programmer manager WEIGHTS 1 3   # 公司决定加薪。。。除了程序员。。
(integer) 6

redis> ZRANGE salary 0 -1 WITHSCORES
1) "peter"
2) "2000"
3) "jack"
4) "3500"
5) "tom"
6) "5000"
7) "herry"
8) "6000"
9) "mary"
10) "10500"
11) "bob"
12) "12000"
```

# ZINTERSTORE

### ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

计算给定的一个或多个有序集的交集，其中给定 `key` 的数量必须以 `numkeys` 参数指定，并将该交集(结果集)储存到 `destination` 。

默认情况下，结果集中某个成员的 `score` 值是所有给定集下该成员 `score` 值之和.

关于 `WEIGHTS` 和 `AGGREGATE` 选项的描述，参见 *ZUNIONSTORE* 命令。

可用版本：

>= 2.0.0

时间复杂度:

O(N*K)+O(M*log(M))， `N` 为给定 `key` 中基数最小的有序集， `K` 为给定有序集的数量， `M` 为结果集的基数。

返回值:

保存到 `destination` 的结果集的基数。

```
redis > ZADD mid_test 70 "Li Lei"
(integer) 1
redis > ZADD mid_test 70 "Han Meimei"
(integer) 1
redis > ZADD mid_test 99.5 "Tom"
(integer) 1

redis > ZADD fin_test 88 "Li Lei"
(integer) 1
redis > ZADD fin_test 75 "Han Meimei"
(integer) 1
redis > ZADD fin_test 99.5 "Tom"
(integer) 1

redis > ZINTERSTORE sum_point 2 mid_test fin_test
(integer) 3

redis > ZRANGE sum_point 0 -1 WITHSCORES     # 显示有序集内所有成员及其 score 值
1) "Han Meimei"
2) "145"
3) "Li Lei"
4) "158"
5) "Tom"
6) "199"
```

# ZSCAN

**ZSCAN key cursor [MATCH pattern] [COUNT count]**

详细信息请参考 *SCAN* 命令。

# Pub/Sub（发布/订阅）

# PSUBSCRIBE

### PSUBSCRIBE pattern [pattern ...]

订阅一个或多个符合给定模式的频道。

每个模式以 `*` 作为匹配符，比如 `it*` 匹配所有以 `it` 开头的频道( `it.news` 、 `it.blog` 、 `it.tweets` 等等)， `news.*` 匹配所有以 `news.` 开头的频道( `news.it` 、 `news.global.today` 等等)，诸如此类。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 是订阅的模式的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 news.* 和 tweet.* 两个模式

# 第 1 - 6 行是执行 psubscribe 之后的反馈信息
# 第 7 - 10 才是接收到的第一条信息
# 第 11 - 14 是第二条
# 以此类推。。。

redis> psubscribe news.* tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"                  # 返回值的类型：显示订阅成功
2) "news.*"                      # 订阅的模式
3) (integer) 1                   # 目前已订阅的模式的数量

1) "psubscribe"
2) "tweet.*"
3) (integer) 2

1) "pmessage"                    # 返回值的类型：信息
2) "news.*"                      # 信息匹配的模式
3) "news.it"                     # 信息本身的目标频道
4) "Google buy Motorola"         # 信息的内容

1) "pmessage"
2) "tweet.*"
3) "tweet.huangz"
4) "hello"

1) "pmessage"
2) "tweet.*"
3) "tweet.joe"
4) "@huangz morning"

1) "pmessage"
2) "news.*"
3) "news.life"
4) "An apple a day, keep doctors away"
```

# PUBLISH

### PUBLISH channel message

将信息 `message` 发送到指定的频道 `channel` 。

可用版本：

>= 2.0.0

时间复杂度：

O(N+M)，其中 `N` 是频道 `channel` 的订阅者数量，而 `M` 则是使用模式订阅(subscribed patterns)的客户端的数量。

返回值：

接收到信息 `message` 的订阅者数量。

```
# 对没有订阅者的频道发送信息

redis> publish bad_channel "can any body hear me?"
(integer) 0

# 向有一个订阅者的频道发送信息

redis> publish msg "good morning"
(integer) 1

# 向有多个订阅者的频道发送信息

redis> publish chat_room "hello~ everyone"
(integer) 3
```

# PUBSUB

**PUBSUB <subcommand> [argument [argument ...]]**

*PUBSUB* 是一个查看订阅与发布系统状态的内省命令， 它由数个不同格式的子命令组成，以下将分别对这些子命令进行介绍。

可用版本： >= 2.8.0

# PUBSUB CHANNELS [pattern]

列出当前的活跃频道。

活跃频道指的是那些至少有一个订阅者的频道， 订阅模式的客户端不计算在内。

`pattern` 参数是可选的：

- 如果不给出 `pattern` 参数，那么列出订阅与发布系统中的所有活跃频道。
- 如果给出 `pattern` 参数，那么只列出和给定模式 `pattern` 相匹配的那些活跃频道。

复杂度： O(N)， `N` 为活跃频道的数量（对于长度较短的频道和模式来说，将进行模式匹配的复杂度视为常数）。

返回值： 一个由活跃频道组成的列表。

```
# client-1 订阅 news.it 和 news.sport 两个频道

client-1> SUBSCRIBE news.it news.sport
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.sport"
3) (integer) 2

# client-2 订阅 news.it 和 news.internet 两个频道

client-2> SUBSCRIBE news.it news.internet
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.internet"
3) (integer) 2

# 首先， client-3 打印所有活跃频道
# 注意，即使一个频道有多个订阅者，它也只输出一次，比如 news.it

client-3> PUBSUB CHANNELS
1) "news.sport"
2) "news.internet"
3) "news.it"

# 接下来， client-3 打印那些与模式 news.i* 相匹配的活跃频道
# 因为 news.sport 不匹配 news.i* ，所以它没有被打印

redis> PUBSUB CHANNELS news.i*
1) "news.internet"
2) "news.it"
```

# PUBSUB NUMSUB [channel-1 ... channel-N]

返回给定频道的订阅者数量， 订阅模式的客户端不计算在内。

复杂度： O(N)， `N` 为给定频道的数量。

返回值： 一个多条批量回复（Multi-bulk reply）， 回复中包含给定的频道，以及频道的订阅者数量。 格式为：频道 `channel-1` ， `channel-1` 的订阅者数量，频道 `channel-2` ， `channel-2` 的订阅者数量，诸如此类。 回复中频道的排列顺序和执行命令时给定频道的排列顺序一致。 不给定任何频道而直接调用这个命令也是可以的， 在这种情况下， 命令只返回一个空列表。

```
# client-1 订阅 news.it 和 news.sport 两个频道

client-1> SUBSCRIBE news.it news.sport
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.sport"
3) (integer) 2

# client-2 订阅 news.it 和 news.internet 两个频道

client-2> SUBSCRIBE news.it news.internet
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "news.it"
3) (integer) 1
1) "subscribe"
2) "news.internet"
3) (integer) 2

# client-3 打印各个频道的订阅者数量

client-3> PUBSUB NUMSUB news.it news.internet news.sport news.music
1) "news.it"    # 频道
2) "2"          # 订阅该频道的客户端数量
3) "news.internet"
4) "1"
5) "news.sport"
6) "1"
7) "news.music" # 没有任何订阅者
8) "0"
```

# PUBSUB NUMPAT

返回订阅模式的数量。

注意， 这个命令返回的不是订阅模式的客户端的数量， 而是客户端订阅的所有模式的数量总和。

复杂度： O(1) 。

返回值： 一个整数回复（Integer reply）。

```
# client-1 订阅 news.* 和 discount.* 两个模式

client-1> PSUBSCRIBE news.* discount.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "news.*"
3) (integer) 1
1) "psubscribe"
2) "discount.*"
3) (integer) 2

# client-2 订阅 tweet.* 一个模式

client-2> PSUBSCRIBE tweet.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "tweet.*"
3) (integer) 1

# client-3 返回当前订阅模式的数量为 3

client-3> PUBSUB NUMPAT
(integer) 3

# 注意，当有多个客户端订阅相同的模式时，相同的订阅也被计算在 PUBSUB NUMPAT 之内
# 比如说，再新建一个客户端 client-4 ，让它也订阅 news.* 频道

client-4> PSUBSCRIBE news.*
Reading messages... (press Ctrl-C to quit)
1) "psubscribe"
2) "news.*"
3) (integer) 1

# 这时再计算被订阅模式的数量，就会得到数量为 4

client-3> PUBSUB NUMPAT
(integer) 4
```

# PUNSUBSCRIBE

**PUNSUBSCRIBE [pattern [pattern ...]]**

指示客户端退订所有给定模式。

如果没有模式被指定，也即是，一个无参数的 `PUNSUBSCRIBE` 调用被执行，那么客户端使用 *PSUBSCRIBE* 命令订阅的所有模式都会被退订。在这种情况下，命令会返回一个信息，告知客户端所有被退订的模式。

可用版本：

>= 2.0.0

时间复杂度：

O(N+M)，其中 `N` 是客户端已订阅的模式的数量， `M` 则是系统中所有客户端订阅的模式的数量。

返回值：

这个命令在不同的客户端中有不同的表现。

# SUBSCRIBE

**SUBSCRIBE channel [channel ...]**

订阅给定的一个或多个频道的信息。

可用版本：

>= 2.0.0

时间复杂度：

O(N)，其中 `N` 是订阅的频道的数量。

返回值：

接收到的信息(请参见下面的代码说明)。

```
# 订阅 msg 和 chat_room 两个频道

# 1 - 6 行是执行 subscribe 之后的反馈信息
# 第 7 - 9 行才是接收到的第一条信息
# 第 10 - 12 行是第二条

redis> subscribe msg chat_room
Reading messages... (press Ctrl-C to quit)
1) "subscribe"       # 返回值的类型：显示订阅成功
2) "msg"             # 订阅的频道名字
3) (integer) 1       # 目前已订阅的频道数量

1) "subscribe"
2) "chat_room"
3) (integer) 2

1) "message"         # 返回值的类型：信息
2) "msg"             # 来源(从那个频道发送过来)
3) "hello moto"      # 信息内容

1) "message"
2) "chat_room"
3) "testing...haha"
```

# UNSUBSCRIBE

**UNSUBSCRIBE [channel [channel ...]]**

指示客户端退订给定的频道。

如果没有频道被指定，也即是，一个无参数的 `UNSUBSCRIBE` 调用被执行，那么客户端使用 *SUBSCRIBE* 命令订阅的所有频道都会被退订。在这种情况下，命令会返回一个信息，告知客户端所有被退订的频道。

可用版本：

>= 2.0.0

时间复杂度：

O(N)， `N` 是客户端已订阅的频道的数量。

返回值：

这个命令在不同的客户端中有不同的表现。

# Transaction（事务）

# DISCARD

### DISCARD

取消事务，放弃执行事务块内的所有命令。

如果正在使用 *WATCH* 命令监视某个(或某些) key，那么取消所有监视，等同于执行命令 *UNWATCH* 。

可用版本：

>= 2.0.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> MULTI
OK

redis> PING
QUEUED

redis> SET greeting "hello"
QUEUED

redis> DISCARD
OK
```

# EXEC

**EXEC**

执行所有事务块内的命令。

假如某个(或某些) key 正处于 *WATCH* 命令的监视之下，且事务块中有和这个(或这些) key 相关的命令，那么 EXEC 命令只在这个(或这些) key 没有被其他命令所改动的情况下执行并生效，否则该事务被打断(abort)。

可用版本：

>= 1.2.0

时间复杂度：

事务块内所有命令的时间复杂度的总和。

返回值：

事务块内所有命令的返回值，按命令执行的先后顺序排列。当操作被打断时，返回空值 `nil` 。

```
# 事务被成功执行

redis> MULTI
OK

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> PING
QUEUED

redis> EXEC
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG

# 监视 key ，且事务成功执行

redis> WATCH lock lock_times
OK

redis> MULTI
OK

redis> SET lock "huangz"
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC
1) OK
2) (integer) 1

# 监视 key ，且事务被打断

redis> WATCH lock lock_times
OK

redis> MULTI
OK

redis> SET lock "joe"        # 就在这时，另一个客户端修改了 lock_times 的值
QUEUED

redis> INCR lock_times
QUEUED

redis> EXEC                  # 因为 lock_times 被修改， joe 的事务执行失败
(nil)
```

# MULTI

### MULTI

标记一个事务块的开始。

事务块内的多条命令会按照先后顺序被放进一个队列当中，最后由 *EXEC* 命令原子性(atomic)地执行。

可用版本：

>= 1.2.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> MULTI            # 标记事务开始
OK

redis> INCR user_id     # 多条命令按顺序入队
QUEUED

redis> INCR user_id
QUEUED

redis> INCR user_id
QUEUED

redis> PING
QUEUED

redis> EXEC             # 执行
1) (integer) 1
2) (integer) 2
3) (integer) 3
4) PONG
```

# UNWATCH

### UNWATCH

取消 *WATCH* 命令对所有 key 的监视。

如果在执行 *WATCH* 命令之后， *EXEC* 命令或 *DISCARD* 命令先被执行了的话，那么就不需要再执行 UNWATCH 了。

因为 *EXEC* 命令会执行事务，因此 *WATCH* 命令的效果已经产生了；而 *DISCARD* 命令在取消事务的同时也会取消所有对 key 的监视，因此这两个命令执行之后，就没有必要执行 UNWATCH 了。

可用版本：

>= 2.2.0

时间复杂度：

O(1)

返回值：

总是 `OK` 。

```
redis> WATCH lock lock_times
OK

redis> UNWATCH
OK
```

# WATCH

### WATCH key [key ...]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

可用版本：

>= 2.2.0

时间复杂度：

O(1)。

返回值：

总是返回 `OK` 。

```
redis> WATCH lock lock_times
OK
```

# Script（脚本）

# EVAL

### EVAL script numkeys key [key ...] arg [arg ...]

从 Redis 2.6.0 版本开始，通过内置的 Lua 解释器，可以使用 EVAL 命令对 Lua 脚本进行求值。

`script` 参数是一段 Lua 5.1 脚本程序，它会被运行在 Redis 服务器上下文中，这段脚本不必(也不应该)定义为一个 Lua 函数。

`numkeys` 参数用于指定键名参数的个数。

键名参数 `key [key ...]` 从 EVAL 的第三个参数开始算起，表示在脚本中所用到的那些 Redis 键(key)，这些键名参数可以在 Lua 中通过全局变量 `KEYS` 数组，用 `1` 为基址的形式访问( `KEYS[1]` ， `KEYS[2]` ，以此类推)。

在命令的最后，那些不是键名参数的附加参数 `arg [arg ...]` ，可以在 Lua 中通过全局变量 `ARGV` 数组访问，访问的形式和 `KEYS` 变量类似( `ARGV[1]` 、 `ARGV[2]` ，诸如此类)。

上面这几段长长的说明可以用一个简单的例子来概括：

```
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

其中 `"return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}"` 是被求值的 Lua 脚本，数字 `2` 指定了键名参数的数量， `key1` 和 `key2` 是键名参数，分别使用 `KEYS[1]` 和 `KEYS[2]` 访问，而最后的 `first` 和 `second` 则是附加参数，可以通过 `ARGV[1]` 和 `ARGV[2]` 访问它们。

在 Lua 脚本中，可以使用两个不同函数来执行 Redis 命令，它们分别是：

- `redis.call()`
- `redis.pcall()`

这两个函数的唯一区别在于它们使用不同的方式处理执行命令所产生的错误，在后面的『错误处理』部分会讲到这一点。

`redis.call()` 和 `redis.pcall()` 两个函数的参数可以是任何格式良好(well formed)的 Redis 命令：

```
> eval "return redis.call('set','foo','bar')" 0
OK
```

需要注意的是，上面这段脚本的确实现了将键 `foo` 的值设为 `bar` 的目的，但是，它违反了 EVAL 命令的语义，因为脚本里使用的所有键都应该由 `KEYS` 数组来传递，就像这样：

```
> eval "return redis.call('set',KEYS[1],'bar')" 1 foo
OK
```

要求使用正确的形式来传递键(key)是有原因的，因为不仅仅是 EVAL 这个命令，所有的 Redis 命令，在执行之前都会被分析，籍此来确定命令会对哪些键进行操作。

因此，对于 EVAL 命令来说，必须使用正确的形式来传递键，才能确保分析工作正确地执行。除此之外，使用正确的形式来传递键还有很多其他好处，它的一个特别重要的用途就是确保 Redis 集群可以将你的请求发送到正确的集群节点。(对 Redis 集群的工作还在进行当中，但是脚本功能被设计成可以与集群功能保持兼容。)不过，这条规矩并不是强制性的，从而使得用户有机会滥用(abuse) Redis 单实例配置(single instance configuration)，代价是这样写出的脚本不能被 Redis 集群所兼容。

# 在 Lua 数据类型和 Redis 数据类型之间转换

当 Lua 通过 `call()` 或 `pcall()` 函数执行 Redis 命令的时候，命令的返回值会被转换成 Lua 数据结构。同样地，当 Lua 脚本在 Redis 内置的解释器里运行时，Lua 脚本的返回值也会被转换成 Redis 协议(protocol)，然后由 EVAL 将值返回给客户端。

数据类型之间的转换遵循这样一个设计原则：如果将一个 Redis 值转换成 Lua 值，之后再将转换所得的 Lua 值转换回 Redis 值，那么这个转换所得的 Redis 值应该和最初时的 Redis 值一样。

换句话说， Lua 类型和 Redis 类型之间存在着一一对应的转换关系。

以下列出的是详细的转换规则：

从 Redis 转换到 Lua ：

- Redis integer reply -> Lua number / Redis 整数转换成 Lua 数字
- Redis bulk reply -> Lua string / Redis bulk 回复转换成 Lua 字符串
- Redis multi bulk reply -> Lua table (may have other Redis data types nested) / Redis 多条 bulk 回复转换成 Lua 表，表内可能有其他别的 Redis 数据类型
- Redis status reply -> Lua table with a single ok field containing the status / Redis 状态回复转换成 Lua 表，表内的 `ok` 域包含了状态信息
- Redis error reply -> Lua table with a single err field containing the error / Redis 错误回复转换成 Lua 表，表内的 `err` 域包含了错误信息
- Redis Nil bulk reply and Nil multi bulk reply -> Lua false boolean type / Redis 的 Nil 回复和 Nil 多条回复转换成 Lua 的布尔值 `false`

从 Lua 转换到 Redis：

- Lua number -> Redis integer reply / Lua 数字转换成 Redis 整数
- Lua string -> Redis bulk reply / Lua 字符串转换成 Redis bulk 回复
- Lua table (array) -> Redis multi bulk reply / Lua 表(数组)转换成 Redis 多条 bulk 回复
- Lua table with a single ok field -> Redis status reply / 一个带单个 `ok` 域的 Lua 表，转换成 Redis 状态回复
- Lua table with a single err field -> Redis error reply / 一个带单个 `err` 域的 Lua 表，转换成 Redis 错误回复
- Lua boolean false -> Redis Nil bulk reply / Lua 的布尔值 `false` 转换成 Redis 的 Nil bulk 回复

从 Lua 转换到 Redis 有一条额外的规则，这条规则没有和它对应的从 Redis 转换到 Lua 的规则：

- Lua boolean true -> Redis integer reply with value of 1 / Lua 布尔值 `true` 转换成 Redis 整数回复中的 `1`

以下是几个类型转换的例子：

```
> eval "return 10" 0
(integer) 10

> eval "return {1,2,{3,'Hello World!'}}" 0
1) (integer) 1
2) (integer) 2
3) 1) (integer) 3
   2) "Hello World!"

> eval "return redis.call('get','foo')" 0
"bar"
```

在上面的三个代码示例里，前两个演示了如何将 Lua 值转换成 Redis 值，最后一个例子更复杂一些，它演示了一个将 Redis 值转换成 Lua 值，然后再将 Lua 值转换成 Redis 值的类型转过程。

# 脚本的原子性

Redis 使用单个 Lua 解释器去运行所有脚本，并且， Redis 也保证脚本会以原子性(atomic)的方式执行：当某个脚本正在运行的时候，不会有其他脚本或 Redis 命令被执行。这和使用 *MULTI* / *EXEC* 包围的事务很类似。在其他别的客户端看来，脚本的效果(effect)要么是不可见的(not visible)，要么就是已完成的(already completed)。

另一方面，这也意味着，执行一个运行缓慢的脚本并不是一个好主意。写一个跑得很快很顺溜的脚本并不难，因为脚本的运行开销(overhead)非常少，但是当你不得不使用一些跑得比较慢的脚本时，请小心，因为当这些蜗牛脚本在慢吞吞地运行的时候，其他客户端会因为服务器正忙而无法执行命令。

# 错误处理

前面的命令介绍部分说过， `redis.call()` 和 `redis.pcall()` 的唯一区别在于它们对错误处理的不同。

当 `redis.call()` 在执行命令的过程中发生错误时，脚本会停止执行，并返回一个脚本错误，错误的输出信息会说明错误造成的原因：

```
redis> lpush foo a
(integer) 1

redis> eval "return redis.call('get', 'foo')" 0
(error) ERR Error running script (call to f_282297a0228f48cd3fc6a55de6316f31422f5d17): ER
```

和 `redis.call()` 不同， `redis.pcall()` 出错时并不引发(raise)错误，而是返回一个带 `err` 域的 Lua 表(table)，用于表示错误：

```
redis 127.0.0.1:6379> EVAL "return redis.pcall('get', 'foo')" 0
(error) ERR Operation against a key holding the wrong kind of value
```

# 带宽和 EVALSHA

EVAL 命令要求你在每次执行脚本的时候都发送一次脚本主体(script body)。Redis 有一个内部的缓存机制，因此它不会每次都重新编译脚本，不过在很多场合，付出无谓的带宽来传送脚本主体并不是最佳选择。

为了减少带宽的消耗， Redis 实现了 EVALSHA 命令，它的作用和 EVAL 一样，都用于对脚本求值，但它接受的第一个参数不是脚本，而是脚本的 SHA1 校验和(sum)。

EVALSHA 命令的表现如下：

- 如果服务器还记得给定的 SHA1 校验和所指定的脚本，那么执行这个脚本
- 如果服务器不记得给定的 SHA1 校验和所指定的脚本，那么它返回一个特殊的错误，提醒用户使用 EVAL 代替 EVALSHA

以下是示例：

```
> set foo bar
OK

> eval "return redis.call('get','foo')" 0
"bar"

> evalsha 6b1bf486c81ceb7edf3c093f4c48582e38c0e791 0
"bar"

> evalsha ffffffffffffffffffffffffffffffffffffffff 0
(error) `NOSCRIPT` No matching script. Please use [EVAL](/commands/eval).
```

客户端库的底层实现可以一直乐观地使用 EVALSHA 来代替 EVAL ，并期望着要使用的脚本已经保存在服务器上了，只有当 `NOSCRIPT` 错误发生时，才使用 EVAL 命令重新发送脚本，这样就可以最大限度地节省带宽。

这也说明了执行 EVAL 命令时，使用正确的格式来传递键名参数和附加参数的重要性：因为如果将参数硬写在脚本中，那么每次当参数改变的时候，都要重新发送脚本，即使脚本的主体并没有改变，相反，通过使用正确的格式来传递键名参数和附加参数，就可以在脚本主体不变的情况下，直接使用 EVALSHA 命令对脚本进行复用，免去了无谓的带宽消耗。

# 脚本缓存

Redis 保证所有被运行过的脚本都会被永久保存在脚本缓存当中，这意味着，当 EVAL 命令在一个 Redis 实例上成功执行某个脚本之后，随后针对这个脚本的所有 EVALSHA 命令都会成功执行。

刷新脚本缓存的唯一办法是显式地调用 `SCRIPT FLUSH` 命令，这个命令会清空运行过的所有脚本的缓存。通常只有在云计算环境中，Redis 实例被改作其他客户或者别的应用程序的实例时，才会执行这个命令。

缓存可以长时间储存而不产生内存问题的原因是，它们的体积非常小，而且数量也非常少，即使脚本在概念上类似于实现一个新命令，即使在一个大规模的程序里有成百上千的脚本，即使这些脚本会经常修改，即便如此，储存这些脚本的内存仍然是微不足道的。

事实上，用户会发现 Redis 不移除缓存中的脚本实际上是一个好主意。比如说，对于一个和 Redis 保持持久化链接(persistent connection)的程序来说，它可以确信，执行过一次的脚本会一直保留在内存当中，因此它可以在流水线中使用 EVALSHA 命令而不必担心因为找不到所需的脚本而产生错误(稍候我们会看到在流水线中执行脚本的相关问题)。

# SCRIPT 命令

Redis 提供了以下几个 SCRIPT 命令，用于对脚本子系统(scripting subsystem)进行控制：

- *SCRIPT FLUSH* ：清除所有脚本缓存
- *SCRIPT EXISTS* ：根据给定的脚本校验和，检查指定的脚本是否存在于脚本缓存
- *SCRIPT LOAD* ：将一个脚本装入脚本缓存，但并不立即运行它
- *SCRIPT KILL* ：杀死当前正在运行的脚本

# 纯函数脚本

在编写脚本方面，一个重要的要求就是，脚本应该被写成纯函数(pure function)。

也就是说，脚本应该具有以下属性：

- 对于同样的数据集输入，给定相同的参数，脚本执行的 Redis 写命令总是相同的。脚本执行的操作不能依赖于任何隐藏(非显式)数据，不能依赖于脚本在执行过程中、或脚本在不同执行时期之间可能变更的状态，并且它也不能依赖于任何来自 I/O 设备的外部输入。

使用系统时间(system time)，调用像 *RANDOMKEY* 那样的随机命令，或者使用 Lua 的随机数生成器，类似以上的这些操作，都会造成脚本的求值无法每次都得出同样的结果。

为了确保脚本符合上面所说的属性， Redis 做了以下工作：

- Lua 没有访问系统时间或者其他内部状态的命令
- Redis 会返回一个错误，阻止这样的脚本运行： 这些脚本在执行随机命令之后(比如 *RANDOMKEY* 、 *SRANDMEMBER* 或 *TIME* 等)，还会执行可以修改数据集的 Redis 命令。如果脚本只是执行只读操作，那么就没有这一限制。注意，随机命令并不一定就指那些带 RAND 字眼的命令，任何带有非确定性的命令都会被认为是随机命令，比如 *TIME* 命令就是这方面的一个很好的例子。
- 每当从 Lua 脚本中调用那些返回无序元素的命令时，执行命令所得的数据在返回给 Lua 之前会先执行一个静默(slient)的字典序排序(lexicographical sorting)。举个例子，因为 Redis 的 Set 保存的是无序的元素，所以在 Redis 命令行客户端中直接执行 *SMEMBERS* ，返回的元素是无序的，但是，假如在脚本中执行 `redis.call("smembers", KEYS[1])` ，那么返回的总是排过序的元素。
- 对 Lua 的伪随机数生成函数 `math.random` 和 `math.randomseed` 进行修改，使得每次在运行新脚本的时候，总是拥有同样的 seed 值。这意味着，每次运行脚本时，只要不使用 `math.randomseed` ，那么 `math.random` 产生的随机数序列总是相同的。

尽管有那么多的限制，但用户还是可以用一个简单的技巧写出带随机行为的脚本(如果他们需要的话)。

假设现在我们要编写一个 Redis 脚本，这个脚本从列表中弹出 N 个随机数。一个 Ruby 写的例子如下：

```
require 'rubygems'
require 'redis'

r = Redis.new

RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,[:mylist],[10,rand(2**32)])
```

这个程序每次运行都会生成带有以下元素的列表：

```
> lrange mylist 0 -1
1) "0.74509509873814"
2) "0.87390407681181"
3) "0.36876626981831"
4) "0.6921941534114"
5) "0.7857992587545"
6) "0.57730350670279"
7) "0.87046522734243"
8) "0.09637165539729"
9) "0.74990198051087"
10) "0.17082803611217"
```

上面的 Ruby 程序每次都只生成同样的列表，用途并不是太大。那么，该怎样修改这个脚本，使得它仍然是一个纯函数(符合 Redis 的要求)，但是每次调用都可以产生不同的随机元素呢？

一个简单的办法是，为脚本添加一个额外的参数，让这个参数作为 Lua 的随机数生成器的 seed 值，这样的话，只要给脚本传入不同的 seed ，脚本就会生成不同的列表元素。

以下是修改后的脚本：

```
RandomPushScript = <<EOF
    local i = tonumber(ARGV[1])
    local res
    math.randomseed(tonumber(ARGV[2]))
    while (i > 0) do
        res = redis.call('lpush',KEYS[1],math.random())
        i = i-1
    end
    return res
EOF

r.del(:mylist)
puts r.eval(RandomPushScript,1,:mylist,10,rand(2**32))
```

尽管对于同样的 seed ，上面的脚本产生的列表元素是一样的(因为它是一个纯函数)，但是只要每次在执行脚本的时候传入不同的 seed ，我们就可以得到带有不同随机元素的列表。

Seed 会在复制(replication link)和写 AOF 文件时作为一个参数来传播，保证在载入 AOF 文件或附属节点(slave)处理脚本时， seed 仍然可以及时得到更新。

注意，Redis 实现保证 `math.random` 和 `math.randomseed` 的输出和运行 Redis 的系统架构无关，无论是 32 位还是 64 位系统，无论是小端(little endian)还是大端(big endian)系统，这两个函数的输出总是相同的。

# 全局变量保护

为了防止不必要的数据泄漏进 Lua 环境， Redis 脚本不允许创建全局变量。如果一个脚本需要在多次执行之间维持某种状态，它应该使用 Redis key 来进行状态保存。

企图在脚本中访问一个全局变量(不论这个变量是否存在)将引起脚本停止， EVAL 命令会返回一个错误：

```
redis 127.0.0.1:6379> eval 'a=10' 0
(error) ERR Error running script (call to f_933044db579a2f8fd45d8065f04a8d0249383e57): us
```

Lua 的 debug 工具，或者其他设施，比如打印（alter）用于实现全局保护的 meta table ，都可以用于实现全局变量保护。

实现全局变量保护并不难，不过有时候还是会不小心而为之。一旦用户在脚本中混入了 Lua 全局状态，那么 AOF 持久化和复制（replication）都会无法保证，所以，请不要使用全局变量。

避免引入全局变量的一个诀窍是：将脚本中用到的所有变量都使用 `local` 关键字定义为局部变量。

## 库

Redis 内置的 Lua 解释器加载了以下 Lua 库：

- `base`
- `table`
- `string`
- `math`
- `debug`
- `cjson`
- `cmsgpack`

其中 `cjson` 库可以让 Lua 以非常快的速度处理 JSON 数据，除此之外，其他别的都是 Lua 的标准库。

每个 Redis 实例都保证会加载上面列举的库，从而确保每个 Redis 脚本的运行环境都是相同的。

# 使用脚本散发 Redis 日志

在 Lua 脚本中，可以通过调用 `redis.log` 函数来写 Redis 日志(log)：

`redis.log(loglevel, message)`

其中， `message` 参数是一个字符串，而 `loglevel` 参数可以是以下任意一个值：

- `redis.LOG_DEBUG`
- `redis.LOG_VERBOSE`
- `redis.LOG_NOTICE`
- `redis.LOG_WARNING`

上面的这些等级(level)和标准 Redis 日志的等级相对应。

对于脚本散发(emit)的日志，只有那些和当前 Redis 实例所设置的日志等级相同或更高级的日志才会被散发。

以下是一个日志示例：

```
redis.log(redis.LOG_WARNING, "Something is wrong with this script.")
```

执行上面的函数会产生这样的信息：

```
[32343] 22 Mar 15:21:39 # Something is wrong with this script.
```

# 沙箱(sandbox)和最大执行时间

脚本应该仅仅用于传递参数和对 Redis 数据进行处理，它不应该尝试去访问外部系统(比如文件系统)，或者执行任何系统调用。

除此之外，脚本还有一个最大执行时间限制，它的默认值是 5 秒钟，一般正常运作的脚本通常可以在几分之几毫秒之内完成，花不了那么多时间，这个限制主要是为了防止因编程错误而造成的无限循环而设置的。

最大执行时间的长短由 `lua-time-limit` 选项来控制(以毫秒为单位)，可以通过编辑 `redis.conf` 文件或者使用 *CONFIG GET* 和 *CONFIG SET* 命令来修改它。

当一个脚本达到最大执行时间的时候，它并不会自动被 Redis 结束，因为 Redis 必须保证脚本执行的原子性，而中途停止脚本的运行意味着可能会留下未处理完的数据在数据集(data set)里面。

因此，当脚本运行的时间超过最大执行时间后，以下动作会被执行：

- Redis 记录一个脚本正在超时运行
- Redis 开始重新接受其他客户端的命令请求，但是只有 `SCRIPT KILL` 和 `SHUTDOWN NOSAVE` 两个命令会被处理，对于其他命令请求， Redis 服务器只是简单地返回 `BUSY` 错误。
- 可以使用 `SCRIPT KILL` 命令将一个仅执行只读命令的脚本杀死，因为只读命令并不修改数据，因此杀死这个脚本并不破坏数据的完整性
- 如果脚本已经执行过写命令，那么唯一允许执行的操作就是 `SHUTDOWN NOSAVE` ，它通过停止服务器来阻止当前数据集写入磁盘

# 流水线(pipeline)上下文(context)中的 EVALSHA

在流水线请求的上下文中使用 EVALSHA 命令时，要特别小心，因为在流水线中，必须保证命令的执行顺序。

一旦在流水线中因为 EVALSHA 命令而发生 NOSCRIPT 错误，那么这个流水线就再也没有办法重新执行了，否则的话，命令的执行顺序就会被打乱。

为了防止出现以上所说的问题，客户端库实现应该实施以下的其中一项措施：

- 总是在流水线中使用 EVAL 命令
- 检查流水线中要用到的所有命令，找到其中的 EVAL 命令，并使用 *SCRIPT EXISTS* 命令检查要用到的脚本是不是全都已经保存在缓存里面了。如果所需的全部脚本都可以在缓存里找到，那么就可以放心地将所有 EVAL 命令改成 EVALSHA 命令，否则的话，就要在流水线的顶端(top)将缺少的脚本用 *SCRIPT LOAD* 命令加上去。

可用版本：

>= 2.6.0

时间复杂度：

EVAL 和 EVALSHA 可以在 O(1) 复杂度内找到要被执行的脚本，其余的复杂度取决于执行的脚本本身。

# EVALSHA

**EVALSHA sha1 numkeys key [key ...] arg [arg ...]**

根据给定的 sha1 校验码，对缓存在服务器中的脚本进行求值。

将脚本缓存到服务器的操作可以通过 *SCRIPT LOAD* 命令进行。

这个命令的其他地方，比如参数的传入方式，都和 *EVAL* 命令一样。

可用版本：

>= 2.6.0

时间复杂度：

根据脚本的复杂度而定。

```
redis> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> EVALSHA "232fd51614574cf0867b83d384a5e898cfd24e5a" 0
"hello moto"
```

# SCRIPT EXISTS

**SCRIPT EXISTS script [script ...]**

给定一个或多个脚本的 SHA1 校验和，返回一个包含 `0` 和 `1` 的列表，表示校验和所指定的脚本是否已经被保存在缓存当中。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N) , `N` 为给定的 SHA1 校验和的数量。

返回值：

一个列表，包含 `0` 和 `1` ，前者表示脚本不存在于缓存，后者表示脚本已经在缓存里面了。列表中的元素和给定的 SHA1 校验和保持对应关系，比如列表的第三个元素的值就表示第三个 SHA1 校验和所指定的脚本在缓存中的状态。

```
redis> SCRIPT LOAD "return 'hello moto'"    # 载入一个脚本
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 1

redis> SCRIPT FLUSH     # 清空缓存
OK

redis> SCRIPT EXISTS 232fd51614574cf0867b83d384a5e898cfd24e5a
1) (integer) 0
```

# SCRIPT FLUSH

### SCRIPT FLUSH

清除所有 Lua 脚本缓存。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

复杂度：

O(N)， `N` 为缓存中脚本的数量。

返回值：

总是返回 `OK`

```
redis> SCRIPT FLUSH
OK
```

# SCRIPT KILL

### SCRIPT KILL

杀死当前正在运行的 Lua 脚本，当且仅当这个脚本没有执行过任何写操作时，这个命令才生效。

这个命令主要用于终止运行时间过长的脚本，比如一个因为 BUG 而发生无限 loop 的脚本，诸如此类。

SCRIPT KILL 执行之后，当前正在运行的脚本会被杀死，执行这个脚本的客户端会从 *EVAL* 命令的阻塞当中退出，并收到一个错误作为返回值。

另一方面，假如当前正在运行的脚本已经执行过写操作，那么即使执行 SCRIPT KILL ，也无法将它杀死，因为这是违反 Lua 脚本的原子性执行原则的。在这种情况下，唯一可行的办法是使用 `SHUTDOWN NOSAVE` 命令，通过停止整个 Redis 进程来停止脚本的运行，并防止不完整(half-written)的信息被写入数据库中。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

执行成功返回 `OK` ，否则返回一个错误。

```
# 没有脚本在执行时

redis> SCRIPT KILL
(error) ERR No scripts in execution right now.

# 成功杀死脚本时

redis> SCRIPT KILL
OK
(1.30s)

# 尝试杀死一个已经执行过写操作的脚本，失败

redis> SCRIPT KILL
(error) ERR Sorry the script already executed write commands against the dataset. You can
(1.69s)
```

以下是脚本被杀死之后，返回给执行脚本的客户端的错误：

```
redis> EVAL "while true do end" 0
(error) ERR Error running script (call to f_694a5fe1ddb97a4c6a1bf299d9537c7d3d0f84e7): Sc
(5.00s)
```

# SCRIPT LOAD

**SCRIPT LOAD script**

将脚本 `script` 添加到脚本缓存中，但并不立即执行这个脚本。

*EVAL* 命令也会将脚本添加到脚本缓存中，但是它会立即对输入的脚本进行求值。

如果给定的脚本已经在缓存里面了，那么不做动作。

在脚本被加入到缓存之后，通过 EVALSHA 命令，可以使用脚本的 SHA1 校验和来调用这个脚本。

脚本可以在缓存中保留无限长的时间，直到执行 *SCRIPT FLUSH* 为止。

关于使用 Redis 对 Lua 脚本进行求值的更多信息，请参见 *EVAL* 命令。

可用版本：

>= 2.6.0

时间复杂度：

O(N) , `N` 为脚本的长度(以字节为单位)。

返回值：

给定 `script` 的 SHA1 校验和

```
redis> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis> EVALSHA 232fd51614574cf0867b83d384a5e898cfd24e5a 0
"hello moto"
```

# Connection（连接）

# AUTH

### AUTH password

通过设置配置文件中 `requirepass` 项的值(使用命令 `CONFIG SET requirepass password` )，可以使用密码来保护 Redis 服务器。

如果开启了密码保护的话，在每次连接 Redis 服务器之后，就要使用 `AUTH` 命令解锁，解锁之后才能使用其他 Redis 命令。

如果 `AUTH` 命令给定的密码 `password` 和配置文件中的密码相符的话，服务器会返回 `OK` 并开始接受命令输入。

另一方面，假如密码不匹配的话，服务器将返回一个错误，并要求客户端需重新输入密码。

Warning

因为 Redis 高性能的特点，在很短时间内尝试猜测非常多个密码是有可能的，因此请确保使用的密码足够复杂和足够长，以免遭受密码猜测攻击。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

密码匹配时返回 `OK` ，否则返回一个错误。

```
# 设置密码

redis> CONFIG SET requirepass secret_password   # 将密码设置为 secret_password
OK

redis> QUIT                                     # 退出再连接，让新密码对客户端生效

[huangz@mypad]$ redis

redis> PING                                     # 未验证密码，操作被拒绝
(error) ERR operation not permitted

redis> AUTH wrong_password_testing              # 尝试输入错误的密码
(error) ERR invalid password

redis> AUTH secret_password                     # 输入正确的密码
OK

redis> PING                                     # 密码验证成功，可以正常操作命令了
PONG

# 清空密码

redis> CONFIG SET requirepass ""   # 通过将密码设为空字符来清空密码
OK

redis> QUIT

$ redis                            # 重新进入客户端

redis> PING                        # 执行命令不再需要密码，清空密码操作成功
PONG
```

# ECHO

### ECHO message

打印一个特定的信息 `message` ，测试时使用。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`message` 自身。

```
redis> ECHO "Hello Moto"
"Hello Moto"

redis> ECHO "Goodbye Moto"
"Goodbye Moto"
```

# PING

### PING

使用客户端向 Redis 服务器发送一个 `PING` ，如果服务器运作正常的话，会返回一个 `PONG` 。

通常用于测试与服务器的连接是否仍然生效，或者用于测量延迟值。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

如果连接正常就返回一个 `PONG` ，否则返回一个连接错误。

```
# 客户端和服务器连接正常

redis> PING
PONG

# 客户端和服务器连接不正常(网络不正常或服务器未能正常运行)

redis 127.0.0.1:6379> PING
Could not connect to Redis at 127.0.0.1:6379: Connection refused
```

# QUIT

**QUIT**

请求服务器关闭与当前客户端的连接。

一旦所有等待中的回复(如果有的话)顺利写入到客户端，连接就会被关闭。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` (但是不会被打印显示，因为当时 Redis-cli 已经退出)。

```
$ redis

redis> QUIT

$
```

# SELECT

### SELECT index

切换到指定的数据库，数据库索引号 `index` 用数字值指定，以 `0` 作为起始索引值。

默认使用 `0` 号数据库。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

`OK`

```
redis> SET db_number 0         # 默认使用 0 号数据库
OK

redis> SELECT 1                # 使用 1 号数据库
OK

redis[1]> GET db_number        # 已经切换到 1 号数据库，注意 Redis 现在的命令提示符多了个 [1]
(nil)

redis[1]> SET db_number 1
OK

redis[1]> GET db_number
"1"

redis[1]> SELECT 3             # 再切换到 3 号数据库
OK

redis[3]>                      # 提示符从 [1] 改变成了 [3]
```

# Server（服务器）

# BGREWRITEAOF

### BGREWRITEAOF

执行一个 AOF文件 重写操作。重写会创建一个当前 AOF 文件的体积优化版本。

即使 BGREWRITEAOF 执行失败，也不会有任何数据丢失，因为旧的 AOF 文件在 BGREWRITEAOF 成功之前不会被修改。

重写操作只会在没有其他持久化工作在后台执行时被触发，也就是说：

- 如果 Redis 的子进程正在执行快照的保存工作，那么 AOF 重写的操作会被预定 (scheduled)，等到保存工作完成之后再执行 AOF 重写。在这种情况下，BGREWRITEAOF 的返回值仍然是 `OK` ，但还会加上一条额外的信息，说明 BGREWRITEAOF 要等到保存操作完成之后才能执行。在 Redis 2.6 或以上的版本，可以使用 *INFO* 命令查看 BGREWRITEAOF 是否被预定。
- 如果已经有别的 AOF 文件重写在执行，那么 BGREWRITEAOF 返回一个错误，并且这个新的 BGREWRITEAOF 请求也不会被预定到下次执行。

从 Redis 2.4 开始， AOF 重写由 Redis 自行触发， BGREWRITEAOF 仅仅用于手动触发重写操作。

请移步 持久化文档(英文) 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要追加到 AOF 文件中的数据数量。

返回值：

反馈信息。

```
redis> BGREWRITEAOF
Background append only file rewriting started
```

# BGSAVE

在后台异步(Asynchronously)保存当前数据库的数据到磁盘。

BGSAVE 命令执行之后立即返回 `OK` ，然后 Redis fork 出一个新子进程，原来的 Redis 进程(父进程)继续处理客户端请求，而子进程则负责将数据保存到磁盘，然后退出。

客户端可以通过 *LASTSAVE* 命令查看相关信息，判断 BGSAVE 命令是否执行成功。

请移步 持久化文档 查看更多相关细节。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要保存到数据库中的 key 的数量。

返回值：

反馈信息。

```
redis> BGSAVE
Background saving started
```

# CLIENT GETNAME

### CLIENT GETNAME

返回 *CLIENT SETNAME* 命令为连接设置的名字。

因为新创建的连接默认是没有名字的， 对于没有名字的连接， *CLIENT GETNAME* 返回空白回复。

可用版本

>= 2.6.9

时间复杂度

O(1)

返回值

如果连接没有设置名字，那么返回空白回复；如果有设置名字，那么返回名字。

```
# 新连接默认没有名字

redis 127.0.0.1:6379> CLIENT GETNAME
(nil)

# 设置名字

redis 127.0.0.1:6379> CLIENT SETNAME hello-world-connection
OK

# 返回名字

redis 127.0.0.1:6379> CLIENT GETNAME
"hello-world-connection"
```

# CLIENT KILL

**CLIENT KILL ip:port**

关闭地址为 `ip:port` 的客户端。

`ip:port` 应该和 *CLIENT LIST* 命令输出的其中一行匹配。

因为 Redis 使用单线程设计，所以当 Redis 正在执行命令的时候，不会有客户端被断开连接。

如果要被断开连接的客户端正在执行命令，那么当这个命令执行之后，在发送下一个命令的时候，它就会收到一个网络错误，告知它自身的连接已被关闭。

可用版本

>= 2.4.0

时间复杂度

O(N)， N 为已连接的客户端数量。

返回值

当指定的客户端存在，且被成功关闭时，返回 `OK` 。

```
# 列出所有已连接客户端

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:43501 fd=5 age=10 idle=0 flags=N db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-fr

# 杀死当前客户端的连接

redis 127.0.0.1:6379> CLIENT KILL 127.0.0.1:43501
OK

# 之前的连接已经被关闭，CLI 客户端又重新建立了连接
# 之前的端口是 43501 ，现在是 43504

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:43504 fd=5 age=0 idle=0 flags=N db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-fre
```

# CLIENT LIST

### CLIENT LIST

以人类可读的格式，返回所有连接到服务器的客户端信息和统计数据。

```
redis> CLIENT LIST
addr=127.0.0.1:43143 fd=6 age=183 idle=0 flags=N db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-f
addr=127.0.0.1:43163 fd=5 age=35 idle=15 flags=N db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-f
addr=127.0.0.1:43167 fd=7 age=24 idle=6 flags=N db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-fr
```

可用版本

>= 2.4.0

时间复杂度

O(N)， N 为连接到服务器的客户端数量。

返回值

命令返回多行字符串，这些字符串按以下形式被格式化：

- 每个已连接客户端对应一行（以 `LF` 分割）
- 每行字符串由一系列 `属性=值` 形式的域组成，每个域之间以空格分开

以下是域的含义：

- `addr` ：客户端的地址和端口
- `fd` ：套接字所使用的文件描述符
- `age` ：以秒计算的已连接时长
- `idle` ：以秒计算的空闲时长
- `flags` ：客户端 flag （见下文）
- `db` ：该客户端正在使用的数据库 ID
- `sub` ：已订阅频道的数量
- `psub` ：已订阅模式的数量
- `multi` ：在事务中被执行的命令数量
- `qbuf` ：查询缓冲区的长度（字节为单位， `0` 表示没有分配查询缓冲区）
- `qbuf-free` ：查询缓冲区剩余空间的长度（字节为单位， `0` 表示没有剩余空间）
- `obl` ：输出缓冲区的长度（字节为单位， `0` 表示没有分配输出缓冲区）
- `oll` ：输出列表包含的对象数量（当输出缓冲区没有剩余空间时，命令回复会以字符串对象的形式被入队到这个队列里）
- `omem` ：输出缓冲区和输出列表占用的内存总量
- `events` ：文件描述符事件（见下文）

- `cmd` ： 最近一次执行的命令

客户端 flag 可以由以下部分组成：

- `O` ： 客户端是 MONITOR 模式下的附属节点（slave）
- `S` ： 客户端是一般模式下（normal）的附属节点
- `M` ： 客户端是主节点（master）
- `x` ： 客户端正在执行事务
- `b` ： 客户端正在等待阻塞事件
- `i` ： 客户端正在等待 VM I/O 操作（已废弃）
- `d` ： 一个受监视（watched）的键已被修改， `EXEC` 命令将失败
- `c` : 在将回复完整地写出之后，关闭链接
- `u` : 客户端未被阻塞（unblocked）
- `A` : 尽可能快地关闭连接
- `N` : 未设置任何 flag

文件描述符事件可以是：

- `r` : 客户端套接字（在事件 loop 中）是可读的（readable）
- `w` : 客户端套接字（在事件 loop 中）是可写的（writeable）

Note

为了 debug 的需要，经常会对域进行添加和删除，一个安全的 Redis 客户端应该可以对 `CLIENT LIST` 的输出进行相应的处理（parse），比如忽略不存在的域，跳过未知域，诸如此类。

# CLIENT SETNAME

**CLIENT SETNAME connection-name**

为当前连接分配一个名字。

这个名字会显示在 *CLIENT LIST* 命令的结果中， 用于识别当前正在与服务器进行连接的客户端。

举个例子， 在使用 Redis 构建队列（queue）时， 可以根据连接负责的任务（role）， 为信息生产者（producer）和信息消费者（consumer）分别设置不同的名字。

名字使用 Redis 的字符串类型来保存， 最大可以占用 512 MB 。 另外， 为了避免和 *CLIENT LIST* 命令的输出格式发生冲突， 名字里不允许使用空格。

要移除一个连接的名字， 可以将连接的名字设为空字符串 `""` 。

使用 *CLIENT GETNAME* 命令可以取出连接的名字。

新创建的连接默认是没有名字的。

Tip

在 Redis 应用程序发生连接泄漏时，为连接设置名字是一种很好的 debug 手段。

可用版本

>= 2.6.9

时间复杂度

O(1)

返回值

设置成功时返回 `OK` 。

```
# 新连接默认没有名字

redis 127.0.0.1:6379> CLIENT GETNAME
(nil)

# 设置名字

redis 127.0.0.1:6379> CLIENT SETNAME hello-world-connection
OK

# 返回名字

redis 127.0.0.1:6379> CLIENT GETNAME
"hello-world-connection"

# 在客户端列表中查看

redis 127.0.0.1:6379> CLIENT LIST
addr=127.0.0.1:36851
fd=5
name=hello-world-connection     # <- 名字
age=51
...

# 清除名字

redis 127.0.0.1:6379> CLIENT SETNAME        # 只用空格是不行的！
(error) ERR Syntax error, try CLIENT (LIST | KILL ip:port)

redis 127.0.0.1:6379> CLIENT SETNAME ""     # 必须双引号显示包围
OK

redis 127.0.0.1:6379> CLIENT GETNAME        # 清除完毕
(nil)
```

# CONFIG GET

### CONFIG GET parameter

CONFIG GET 命令用于取得运行中的 Redis 服务器的配置参数(configuration parameters)，在 Redis 2.4 版本中， 有部分参数没有办法用 `CONFIG GET` 访问，但是在最新的 Redis 2.6 版本中，所有配置参数都已经可以用 `CONFIG GET` 访问了。

CONFIG GET 接受单个参数 `parameter` 作为搜索关键字，查找所有匹配的配置参数，其中参数和值以“键-值对”(key-value pairs)的方式排列。

比如执行 `CONFIG GET s*` 命令，服务器就会返回所有以 `s` 开头的配置参数及参数的值：

```
redis> CONFIG GET s*
1) "save"                       # 参数名：save
2) "900 1 300 10 60 10000"      # save 参数的值
3) "slave-serve-stale-data"     # 参数名： slave-serve-stale-data
4) "yes"                        # slave-serve-stale-data 参数的值
5) "set-max-intset-entries"     # ...
6) "512"
7) "slowlog-log-slower-than"
8) "1000"
9) "slowlog-max-len"
10) "1000"
```

如果你只是寻找特定的某个参数的话，你当然也可以直接指定参数的名字：

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"
```

使用命令 `CONFIG GET *` ， 可以列出 `CONFIG GET` 命令支持的所有参数：

```
redis> CONFIG GET *
1) "dir"
2) "/var/lib/redis"
3) "dbfilename"
4) "dump.rdb"
5) "requirepass"
6) (nil)
7) "masterauth"
8) (nil)
9) "maxmemory"
10) "0"
11) "maxmemory-policy"
12) "volatile-lru"
13) "maxmemory-samples"
14) "3"
15) "timeout"
16) "0"
17) "appendonly"
18) "no"
# ...
49) "loglevel"
50) "verbose"
```

所有被 `CONFIG SET` 所支持的配置参数都可以在配置文件 redis.conf 中找到，不过 `CONFIG GET` 和 `CONFIG SET` 使用的格式和 redis.conf 文件所使用的格式有以下两点不同：

- `10kb` 、 `2gb` 这些在配置文件中所使用的储存单位缩写，不可以用在 `CONFIG` 命令中， `CONFIG SET` 的值只能通过数字值显式地设定。像 `CONFIG SET xxx 1k` 这样的命令是错误的，正确的格式是 `CONFIG SET xxx 1000` 。
- `save` 选项在 redis.conf 中是用多行文字储存的，但在 `CONFIG GET` 命令中，它只打印一行文字。以下是 `save` 选项在 redis.conf 文件中的表示：```save 900 1``save 300 10``save 60 10000```但是 `CONFIG GET` 命令的输出只有一行：```redis> CONFIG GET save``1) "save"``2) "900 1 300 10 60 10000"```上面 `save` 参数的三个值表示：在 900 秒内最少有 1 个 key 被改动，或者 300 秒内最少有 10 个 key 被改动，又或者 60 秒内最少有 1000 个 key 被改动，以上三个条件随便满足一个，就触发一次保存操作。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

给定配置参数的值。

# CONFIG RESETSTAT

### CONFIG RESETSTAT

重置 *INFO* 命令中的某些统计数据，包括：

- Keyspace hits (键空间命中次数)
- Keyspace misses (键空间不命中次数)
- Number of commands processed (执行命令的次数)
- Number of connections received (连接服务器的次数)
- Number of expired keys (过期key的数量)
- Number of rejected connections (被拒绝的连接数量)
- Latest fork(2) time(最后执行 fork(2) 的时间)
- The `aof_delayed_fsync` counter( `aof_delayed_fsync` 计数器的值)

可用版本：

>= 2.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` 。

```
# 重置前

redis 127.0.0.1:6379> INFO
# Server
redis_version:2.5.3
redis_git_sha1:d0407c2d
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
gcc_version:4.6.3
process_id:11095
run_id:ef1f6b6c7392e52d6001eaf777acbe547d1192e2
tcp_port:6379
uptime_in_seconds:6
uptime_in_days:0
lru_clock:1205426

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:331076
used_memory_human:323.32K
used_memory_rss:1568768
used_memory_peak:293424
```

```
used_memory_peak_human:286.55K
used_memory_lua:16384
mem_fragmentation_ratio:4.74
mem_allocator:jemalloc-2.2.5

# Persistence
loading:0
aof_enabled:0
changes_since_last_save:0
bgsave_in_progress:0
last_save_time:1333260015
last_bgsave_status:ok
bgrewriteaof_in_progress:0

# Stats
total_connections_received:1
total_commands_processed:0
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
connected_slaves:0

# CPU
used_cpu_sys:0.01
used_cpu_user:0.00
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
db0:keys=20,expires=0

# 重置

redis 127.0.0.1:6379> CONFIG RESETSTAT
OK

# 重置后

redis 127.0.0.1:6379> INFO
# Server
redis_version:2.5.3
redis_git_sha1:d0407c2d
redis_git_dirty:0
arch_bits:32
multiplexing_api:epoll
gcc_version:4.6.3
process_id:11095
run_id:ef1f6b6c7392e52d6001eaf777acbe547d1192e2
tcp_port:6379
uptime_in_seconds:134
uptime_in_days:0
lru_clock:1205438

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:331076
used_memory_human:323.32K
used_memory_rss:1568768
```

```
used_memory_peak:330280
used_memory_peak_human:322.54K
used_memory_lua:16384
mem_fragmentation_ratio:4.74
mem_allocator:jemalloc-2.2.5

# Persistence
loading:0
aof_enabled:0
changes_since_last_save:0
bgsave_in_progress:0
last_save_time:1333260015
last_bgsave_status:ok
bgrewriteaof_in_progress:0

# Stats
total_connections_received:0
total_commands_processed:1
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
connected_slaves:0

# CPU
used_cpu_sys:0.05
used_cpu_user:0.02
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
db0:keys=20,expires=0
```

# CONFIG REWRITE

### CONFIG REWRITE

*CONFIG REWRITE* 命令对启动 Redis 服务器时所指定的 `redis.conf` 文件进行改写： 因为 *CONFIG SET* 命令可以对服务器的当前配置进行修改， 而修改后的配置可能和 `redis.conf` 文件中所描述的配置不一样， *CONFIG REWRITE* 的作用就是通过尽可能少的修改， 将服务器当前所使用的配置记录到 `redis.conf` 文件中。

重写会以非常保守的方式进行：

- 原有 `redis.conf` 文件的整体结构和注释会被尽可能地保留。
- 如果一个选项已经存在于原有 `redis.conf` 文件中， 那么对该选项的重写会在选项原本所在的位置（行号）上进行。
- 如果一个选项不存在于原有 `redis.conf` 文件中， 并且该选项被设置为默认值， 那么重写程序不会将这个选项添加到重写后的 `redis.conf` 文件中。
- 如果一个选项不存在于原有 `redis.conf` 文件中， 并且该选项被设置为非默认值， 那么这个选项将被添加到重写后的 `redis.conf` 文件的末尾。
- 未使用的行会被留白。 比如说， 如果你在原有 `redis.conf` 文件上设置了数个关于 `save` 选项的参数， 但现在你将这些 `save` 参数的一个或全部都关闭了， 那么这些不再使用的参数原本所在的行就会变成空白的。

即使启动服务器时所指定的 `redis.conf` 文件已经不再存在， *CONFIG REWRITE* 命令也可以重新构建并生成出一个新的 `redis.conf` 文件。

另一方面， 如果启动服务器时没有载入 `redis.conf` 文件， 那么执行 *CONFIG REWRITE* 命令将引发一个错误。

## 原子性重写

对 `redis.conf` 文件的重写是原子性的， 并且是一致的： 如果重写出错或重写期间服务器崩溃， 那么重写失败， 原有 `redis.conf` 文件不会被修改。 如果重写成功， 那么 `redis.conf` 文件为重写后的新文件。

## 可用版本

>= 2.8.0

## 返回值

一个状态值：如果配置重写成功则返回 `OK` ，失败则返回一个错误。

# 测试

以下是执行 *CONFIG REWRITE* 前， 被载入到 Redis 服务器的 `redis.conf` 文件中关于 `appendonly` 选项的设置：

```
# ... 其他选项

appendonly no

# ... 其他选项
```

在执行以下命令之后：

```
127.0.0.1:6379> CONFIG GET appendonly           # appendonly 处于关闭状态
1) "appendonly"
2) "no"

127.0.0.1:6379> CONFIG SET appendonly yes       # 打开 appendonly
OK

127.0.0.1:6379> CONFIG GET appendonly
1) "appendonly"
2) "yes"

127.0.0.1:6379> CONFIG REWRITE                  # 将 appendonly 的修改写入到 redis.conf 中
OK
```

重写后的 `redis.conf` 文件中的 `appendonly` 选项将被改写：

```
# ... 其他选项

appendonly yes

# ... 其他选项
```

# CONFIG SET

**CONFIG SET parameter value**

CONFIG SET 命令可以动态地调整 Redis 服务器的配置(configuration)而无须重启。

你可以使用它修改配置参数，或者改变 Redis 的持久化(Persistence)方式。

CONFIG SET 可以修改的配置参数可以使用命令 `CONFIG GET *` 来列出，所有被 CONFIG SET 修改的配置参数都会立即生效。

关于 CONFIG SET 命令的更多消息，请参见命令 *CONFIG GET* 的说明。

关于如何使用 CONFIG SET 命令修改 Redis 持久化方式，请参见 Redis Persistence 。

可用版本：

>= 2.0.0

时间复杂度：

不明确

返回值：

当设置成功时返回 `OK` ，否则返回一个错误。

```
redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1024"

redis> CONFIG SET slowlog-max-len 10086
OK

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "10086"
```

# DBSIZE

### DBSIZE

返回当前数据库的 key 的数量。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当前数据库的 key 的数量。

```
redis> DBSIZE
(integer) 5

redis> SET new_key "hello_moto"     # 增加一个 key 试试
OK

redis> DBSIZE
(integer) 6
```

# DEBUG OBJECT

### DEBUG OBJECT key

DEBUG OBJECT 是一个调试命令，它不应被客户端所使用。

查看 *OBJECT* 命令获取更多信息。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

当 `key` 存在时，返回有关信息。当 `key` 不存在时，返回一个错误。

```
redis> DEBUG OBJECT my_pc
Value at:0xb6838d20 refcount:1 encoding:raw serializedlength:9 lru:283790 lru_seconds_idl

redis> DEBUG OBJECT your_mac
(error) ERR no such key
```

# DEBUG SEGFAULT

### DEBUG SEGFAULT

执行一个不合法的内存访问从而让 Redis 崩溃，仅在开发时用于 BUG 模拟。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

无

```
redis> DEBUG SEGFAULT
Could not connect to Redis at: Connection refused

not connected>
```

# FLUSHALL

### FLUSHALL

清空整个 Redis 服务器的数据(删除所有数据库的所有 key )。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

尚未明确

返回值：

总是返回 `OK` 。

```
redis> DBSIZE            # 0 号数据库的 key 数量
(integer) 9

redis> SELECT 1          # 切换到 1 号数据库
OK

redis[1]> DBSIZE         # 1 号数据库的 key 数量
(integer) 6

redis[1]> flushall       # 清空所有数据库的所有 key
OK

redis[1]> DBSIZE         # 不但 1 号数据库被清空了
(integer) 0

redis[1]> SELECT 0       # 0 号数据库(以及其他所有数据库)也一样
OK

redis> DBSIZE
(integer) 0
```

# FLUSHDB

**FLUSHDB**

清空当前数据库中的所有 key。

此命令从不失败。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

总是返回 `OK` 。

```
redis> DBSIZE    # 清空前的 key 数量
(integer) 4

redis> FLUSHDB
OK

redis> DBSIZE    # 清空后的 key 数量
(integer) 0
```

# INFO

**INFO [section]**

以一种易于解释（parse）且易于阅读的格式，返回关于 Redis 服务器的各种信息和统计数值。

通过给定可选的参数 `section` ，可以让命令只返回某一部分的信息：

- `server` : 一般 Redis 服务器信息，包含以下域：

  > `redis_version` : *Redis 服务器版本* > `redis_git_sha1` : Git SHA1 > `redis_git_dirty` : *Git dirty flag* > `os` : Redis 服务器的宿主操作系统 > `arch_bits` : *架构（32 或 64 位）* > `multiplexing_api` : Redis 所使用的事件处理机制 > `gcc_version` : *编译 Redis 时所使用的 GCC 版本* > `process_id` : 服务器进程的 PID > `run_id` : *Redis 服务器的随机标识符（用于 Sentinel 和集群）* > `tcp_port` : TCP/IP 监听端口 > `uptime_in_seconds` : *自 Redis 服务器启动以来，经过的秒数* > `uptime_in_days` : 自 Redis 服务器启动以来，经过的天数 > * `lru_clock` : 以分钟为单位进行自增的时钟，用于 LRU 管理

- `clients` : 已连接客户端信息，包含以下域：

  > `connected_clients` : *已连接客户端的数量（不包括通过从属服务器连接的客户端）* > `client_longest_output_list` : 当前连接的客户端当中，最长的输出列表 > `client_longest_input_buf` : *当前连接的客户端当中，最大输入缓存* > `blocked_clients` : 正在等待阻塞命令（BLPOP、BRPOP、BRPOPLPUSH）的客户端的数量

- `memory` : 内存信息，包含以下域：

  > `used_memory` : *由 Redis 分配器分配的内存总量，以字节（byte）为单位* > `used_memory_human` : 以人类可读的格式返回 Redis 分配的内存总量 > `used_memory_rss` : *从操作系统的角度，返回 Redis 已分配的内存总量（俗称常驻集大小）。这个值和 `top` 、 `ps` 等命令的输出一致。* > `used_memory_peak` : Redis 的内存消耗峰值（以字节为单位） > `used_memory_peak_human` : *以人类可读的格式返回 Redis 的内存消耗峰值* > `used_memory_lua` : Lua 引擎所使用的内存大小（以字节为单位） > `mem_fragmentation_ratio` : *`used_memory_rss` 和 `used_memory` 之间的比率* > `mem_allocator` : 在编译时指定的， Redis 所使用的内存分配器。可以是 libc 、 jemalloc 或者 tcmalloc 。 > > 在理想情况下， `used_memory_rss` 的值应该只比 `used_memory` 稍微高一点儿。当 `rss > used` ，且两者的值相差较大时，表示存在（内部或外部的）内存碎片。内存碎片的比率可以通过 `mem_fragmentation_ratio` 的值看出。当 `used > rss` 时，表示 Redis 的部分内存被操作系统换出到交换空间了，在这种情况下，操作可能会产生明显的延迟。 > > Because Redis does not have control over how its allocations are mapped to memory pages, high `used_memory_rss` is often the result

of a spike in memory usage. > > 当 Redis 释放内存时，分配器可能会，也可能不会，将内存返还给操作系统。如果 Redis 释放了内存，却没有将内存返还给操作系统，那么 `used_memory` 的值可能和操作系统显示的 Redis 内存占用并不一致。查看 `used_memory_peak` 的值可以验证这种情况是否发生。

- `persistence` : `RDB` 和 `AOF` 的相关信息
- `stats` : 一般统计信息
- `replication` : 主/从复制信息
- `cpu` : CPU 计算量统计信息
- `commandstats` : Redis 命令统计信息
- `cluster` : Redis 集群信息
- `keyspace` : 数据库相关的统计信息

除上面给出的这些值以外，参数还可以是下面这两个：

- `all` : 返回所有信息
- `default` : 返回默认选择的信息

当不带参数直接调用 INFO 命令时，使用 `default` 作为默认参数。

Note

不同版本的 Redis 可能对返回的一些域进行了增加或删减。

因此，一个健壮的客户端程序在对 *INFO* 命令的输出进行分析时，应该能够跳过不认识的域，并且妥善地处理丢失不见的域。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

具体请参见下面的测试代码。

```
redis> INFO
# Server
redis_version:2.5.9
redis_git_sha1:473f3090
redis_git_dirty:0
os:Linux 3.3.7-1-ARCH i686
arch_bits:32
multiplexing_api:epoll
gcc_version:4.7.0
process_id:8104
run_id:bc9e20c6f0aac67d0d396ab950940ae4d1479ad1
tcp_port:6379
uptime_in_seconds:7
uptime_in_days:0
lru_clock:1680564

# Clients
connected_clients:1
client_longest_output_list:0
client_biggest_input_buf:0
blocked_clients:0

# Memory
used_memory:439304
used_memory_human:429.01K
used_memory_rss:13897728
used_memory_peak:401776
used_memory_peak_human:392.36K
used_memory_lua:20480
mem_fragmentation_ratio:31.64
mem_allocator:jemalloc-3.0.0

# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1338011402
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:-1
rdb_current_bgsave_time_sec:-1
aof_enabled:0
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1

# Stats
total_connections_received:1
total_commands_processed:0
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0

# Replication
role:master
connected_slaves:0

# CPU
used_cpu_sys:0.03
used_cpu_user:0.01
used_cpu_sys_children:0.00
used_cpu_user_children:0.00

# Keyspace
```

# LASTSAVE

### LASTSAVE

返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示。

可用版本：

>= 1.0.0

时间复杂度：

O(1)

返回值：

一个 UNIX 时间戳。

```
redis> LASTSAVE
(integer) 1324043588
```

# MONITOR

### MONITOR

实时打印出 Redis 服务器接收到的命令，调试用。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

总是返回 `OK` 。

```
127.0.0.1:6379> MONITOR
OK
# 以第一个打印值为例
# 1378822099.421623 是时间戳
# [0 127.0.0.1:56604] 中的 0 是数据库号码， 127... 是 IP 地址和端口
# "PING" 是被执行的命令
1378822099.421623 [0 127.0.0.1:56604] "PING"
1378822105.089572 [0 127.0.0.1:56604] "SET" "msg" "hello world"
1378822109.036925 [0 127.0.0.1:56604] "SET" "number" "123"
1378822140.649496 [0 127.0.0.1:56604] "SADD" "fruits" "Apple" "Banana" "Cherry"
1378822154.117160 [0 127.0.0.1:56604] "EXPIRE" "msg" "10086"
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
1378822258.690131 [0 127.0.0.1:56604] "DBSIZE"
```

# PSYNC

### PSYNC <MASTER_RUN_ID> <OFFSET>

用于复制功能(replication)的内部命令。

更多信息请参考 复制（*Replication*） 文档。

可用版本：

>= 2.8.0

时间复杂度：

不明确

返回值：

不明确

```
127.0.0.1:6379> PSYNC ? -1
"REDIS0006\xfe\x00\x00\x02kk\x02vv\x00\x03msg\x05hello\xff\xc3\x96P\x12h\bK\xef"
```

# SAVE

### SAVE

SAVE 命令执行一个同步保存操作，将当前 Redis 实例的所有数据快照(snapshot)以 RDB 文件的形式保存到硬盘。

一般来说，在生产环境很少执行 SAVE 操作，因为它会阻塞所有客户端，保存数据库的任务通常由 *BGSAVE* 命令异步地执行。然而，如果负责保存数据的后台子进程不幸出现问题时，SAVE 可以作为保存数据的最后手段来使用。

请参考文档： Redis 的持久化运作方式(英文) 以获取更多消息。

可用版本：

>= 1.0.0

时间复杂度：

O(N)， `N` 为要保存到数据库中的 key 的数量。

返回值：

保存成功时返回 `OK` 。

```
redis> SAVE
OK
```

# SHUTDOWN

### SHUTDOWN

SHUTDOWN 命令执行以下操作：

- 停止所有客户端
- 如果有至少一个保存点在等待，执行 *SAVE* 命令
- 如果 AOF 选项被打开，更新 AOF 文件
- 关闭 redis 服务器(server)

如果持久化被打开的话， SHUTDOWN 命令会保证服务器正常关闭而不丢失任何数据。

另一方面，假如只是单纯地执行 *SAVE* 命令，然后再执行 *QUIT* 命令，则没有这一保证 —— 因为在执行 *SAVE* 之后、执行 *QUIT* 之前的这段时间中间，其他客户端可能正在和服务器进行通讯，这时如果执行 *QUIT* 就会造成数据丢失。

### SAVE 和 NOSAVE 修饰符

通过使用可选的修饰符，可以修改 SHUTDOWN 命令的表现。比如说：

- 执行 `SHUTDOWN SAVE` 会强制让数据库执行保存操作，即使没有设定(configure)保存点
- 执行 `SHUTDOWN NOSAVE` 会阻止数据库执行保存操作，即使已经设定有一个或多个保存点(你可以将这一用法看作是强制停止服务器的一个假想的 ABORT 命令)

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

执行失败时返回错误。执行成功时不返回任何信息，服务器和客户端的连接断开，客户端自动退出。

```
redis> PING
PONG

redis> SHUTDOWN

$

$ redis
Could not connect to Redis at: Connection refused
not connected>
```

# SLAVEOF

`SLAVEOF host port`

SLAVEOF 命令用于在 Redis 运行时动态地修改复制(replication)功能的行为。

通过执行 `SLAVEOF host port` 命令，可以将当前服务器转变为指定服务器的从属服务器(slave server)。

如果当前服务器已经是某个主服务器(master server)的从属服务器，那么执行 `SLAVEOF host port` 将使当前服务器停止对旧主服务器的同步，丢弃旧数据集，转而开始对新主服务器进行同步。

另外，对一个从属服务器执行命令 `SLAVEOF NO ONE` 将使得这个从属服务器关闭复制功能，并从从属服务器转变回主服务器，原来同步所得的数据集不会被丢弃。

利用『 `SLAVEOF NO ONE` 不会丢弃同步所得数据集』这个特性，可以在主服务器失败的时候，将从属服务器用作新的主服务器，从而实现无间断运行。

可用版本：

>= 1.0.0

时间复杂度：

`SLAVEOF host port` ，O(N)， `N` 为要同步的数据数量。`SLAVEOF NO ONE` ， O(1) 。

返回值：

总是返回 `OK` 。

```
redis> SLAVEOF 127.0.0.1 6379
OK

redis> SLAVEOF NO ONE
OK
```

# SLOWLOG

**SLOWLOG subcommand [argument]**

什么是 **SLOWLOG**

Slow log 是 Redis 用来记录查询执行时间的日志系统。

查询执行时间指的是不包括像客户端响应(talking)、发送回复等 IO 操作，而单单是执行一个查询命令所耗费的时间。

另外，slow log 保存在内存里面，读写速度非常快，因此你可以放心地使用它，不必担心因为开启 slow log 而损害 Redis 的速度。

设置 **SLOWLOG**

Slow log 的行为由两个配置参数(configuration parameter)指定，可以通过改写 redis.conf 文件或者用 `CONFIG GET` 和 `CONFIG SET` 命令对它们动态地进行修改。

第一个选项是 `slowlog-log-slower-than` ，它决定要对执行时间大于多少微秒(microsecond，1秒 = 1,000,000 微秒)的查询进行记录。

比如执行以下命令将让 slow log 记录所有查询时间大于等于 100 微秒的查询：

`CONFIG SET slowlog-log-slower-than 100`

而以下命令记录所有查询时间大于 1000 微秒的查询：

`CONFIG SET slowlog-log-slower-than 1000`

另一个选项是 `slowlog-max-len` ，它决定 slow log 最多能保存多少条日志， slow log 本身是一个 FIFO 队列，当队列大小超过 `slowlog-max-len` 时，最旧的一条日志将被删除，而最新的一条日志加入到 slow log ，以此类推。

以下命令让 slow log 最多保存 1000 条日志：

`CONFIG SET slowlog-max-len 1000`

使用 `CONFIG GET` 命令可以查询两个选项的当前值：

```
redis> CONFIG GET slowlog-log-slower-than
1) "slowlog-log-slower-than"
2) "1000"

redis> CONFIG GET slowlog-max-len
1) "slowlog-max-len"
2) "1000"
```

查看 **slow log**

要查看 slow log，可以使用 `SLOWLOG GET` 或者 `SLOWLOG GET number` 命令，前者打印所有 slow log，最大长度取决于 `slowlog-max-len` 选项的值，而 `SLOWLOG GET number` 则只打印指定数量的日志。

最新的日志会最先被打印：

```
# 为测试需要，将 slowlog-log-slower-than 设成了 10 微秒

redis> SLOWLOG GET
1) 1) (integer) 12                      # 唯一性(unique)的日志标识符
   2) (integer) 1324097834              # 被记录命令的执行时间点，以 UNIX 时间戳格式表示
   3) (integer) 16                      # 查询执行时间，以微秒为单位
   4) 1) "CONFIG"                       # 执行的命令，以数组的形式排列
      2) "GET"                          # 这里完整的命令是 CONFIG GET slowlog-log-slower-tha
      3) "slowlog-log-slower-than"

2) 1) (integer) 11
   2) (integer) 1324097825
   3) (integer) 42
   4) 1) "CONFIG"
      2) "GET"
      3) "*"

3) 1) (integer) 10
   2) (integer) 1324097820
   3) (integer) 11
   4) 1) "CONFIG"
      2) "GET"
      3) "slowlog-log-slower-than"

# ...
```

日志的唯一 id 只有在 Redis 服务器重启的时候才会重置，这样可以避免对日志的重复处理(比如你可能会想在每次发现新的慢查询时发邮件通知你)。

查看当前日志的数量

使用命令 `SLOWLOG LEN` 可以查看当前日志的数量。

请注意这个值和 `slower-max-len` 的区别，它们一个是当前日志的数量，一个是允许记录的最大日志的数量。

```
redis> SLOWLOG LEN
(integer) 14
```

清空日志

使用命令 `SLOWLOG RESET` 可以清空 slow log 。

```
redis> SLOWLOG LEN
(integer) 14

redis> SLOWLOG RESET
OK

redis> SLOWLOG LEN
(integer) 0
```

可用版本：

>= 2.2.12

时间复杂度：

O(1)

返回值：

取决于不同命令，返回不同的值。

# SYNC

### SYNC

用于复制功能(replication)的内部命令。

更多信息请参考 Redis 官网的 Replication 章节 。

可用版本：

>= 1.0.0

时间复杂度：

不明确

返回值：

不明确

```
redis> SYNC
"REDIS0002\xfe\x00\x00\auser_id\xc0\x03\x00\anumbers\xc2\xf3\xe0\x01\x00\x00\tdb_number\x
(1.90s)
```

# TIME

### TIME

返回当前服务器时间。

可用版本：

>= 2.6.0

时间复杂度：

O(1)

返回值：

一个包含两个字符串的列表： 第一个字符串是当前时间(以 UNIX 时间戳格式表示)，而第二个字符串是当前这一秒钟已经逝去的微秒数。

```
redis> TIME
1) "1332395997"
2) "952581"
redis> TIME
1) "1332395997"
2) "953148"
```

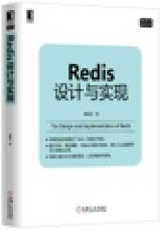
# 关于

本文档由 黄健宏（huangz） 翻译， 版权归 Redis 官方所有。

更新日志(change log) 列出了本文档的主要更新细节， 你也可以通过关注 文档的 github 项目 来随时追踪文档的最新更新信息。

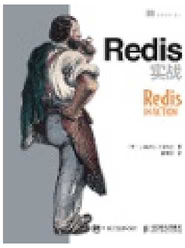
有任何问题、意见或建议， 请在文档配套的 disqus 论坛里留言， 或者直接联系译者。

## Redis 书籍推荐



由本文档译者黄健宏创作的《Redis 设计与实现》一书正在销售中， 该书详细地介绍了 Redis 内部的运作原理以及各项功能的实现原理， 是一本致力于帮助 Redis 使用者加深对 Redis 的理解， 并且更高效地使用 Redis 的书籍。

欢迎访问 RedisBook.com 并了解《Redis 设计与实现》的更多相关信息。



由《Redis命令参考》的译者黄健宏翻译的《Redis实战》一书正在火热发售中， 该书深入浅出地介绍了 Redis 的五种数据结构， 并通过一系列实用的示例深刻地展示了 Redis 的用法。 此外， 《Redis实战》还介绍了多种扩展和优化 Redis 的方法， 无论是 Redis 新手还是有一定经验的 Redis 使用者， 应该都能从此书中获益。

欢迎访问 redisinaction.com 并了解《Redis实战》的更多相关信息。

## 参加群讨论

欢迎各位《Redis命令参考》读者加入 Redis 技术讨论 QQ 群 398976550， 你可以在群里面分享你的 Redis 使用心得， 又或者跟其他人讨论你在使用 Redis 过程中遇到的问题。